

LABORATOIRE



INFORMATIQUE, SIGNAUX ET SYSTÈMES
DE SOPHIA ANTIPOLIS
UMR 6070

TOWARDS DOMAIN-DRIVEN DEVELOPMENT: APPROACH AND IMPLEMENTATION

Carine Courbis, Philippe Lahire, Didier Parigot

Projet OCL

Rapport de recherche
ISRN I3S/RR-2004-14-FR

Avril 2004

RÉSUMÉ :

Avec Internet et la prolifération des nouvelles technologies de composants et de distribution, la conception et l'implémentation des applications complexes doit s'approcher des standards, du code distribué, du déploiement de composants et des savoir-faire métiers. Pour affronter tous ces changements, les applications ont besoin d'être plus ouvertes, adaptables et capable d'évoluer.

Pour résoudre des nouveaux défis, ce rapport présente une nouvelle approche de développement basée sur les transformations de différents modèles métiers, chacun d'eux étant en relation avec une préoccupation de l'application.

MOTS CLÉS :

Développement dirigés par les domaines, méta-modélisation, programmation générative, séparation des préoccupations

ABSTRACT:

With the Internet and the proliferation of new component and distributive technologies, the design and implementation of complex applications must take into account standards, code distribution, deployment of components and reuse of business know-hows. To cope with these changes, applications need to be more open, adaptable and capable of evolving.

To accommodate to these new challenges, this paper presents a new development approach based on transformations of different business models, each of them related to one possible concern of the application.

KEY WORDS :

Domain-Driven Development, Meta-Modelling, Generative Programming, Separation of Concerns

Towards Domain-Driven Development: Approach and Implementation

Didier Parigot¹, Philippe Lahire², and Carine Courbis³

¹ INRIA Sophia-Antipolis - 2004 route des Lucioles - BP 93
F-06902 Sophia-Antipolis cedex - France

`Didier.Parigot@inria.fr`

`http://www.inria.fr/`

² Laboratoire I3S (UNSA/CNRS) - 2000 route des Lucioles - BP 121 -
F-06903 Sophia-Antipolis cedex - France

`Philippe.Lahire@unice.fr`

`http://www.i3s.unice.fr/`

³ University College London - Computer Science Department - Adastral Park,
Martlesham IP5 3RE - United Kingdom

`Carine.Courbis@bt.com` `http://www.cs.ucl.ac.uk/`

Abstract. With the Internet and the proliferation of new component and distributive technologies, the design and implementation of complex applications must take into account standards, code distribution, deployment of components and reuse of business know-hows. To cope with these changes, applications need to be more open, adaptable and capable of evolving.

To accommodate to these new challenges, this paper presents a new development approach based on transformations of different business models, each of them related to one possible concern of the application.

This approach is MDA compliant. It relies on Generative and Component Programming and on approaches by Separation of Concerns (ASoC) which are adapted for business-model descriptions. It contributes to the research works on Domain Driven Development and have a fully operational implementation (SMARTTOOLS). It may be compared to a software factory dedicated to applications which rely on a data model.

The main expected results are i) to build software of better quality thanks to business models and technology separation, ii) to generate simpler code, iii) to enable rapid developments and insertions of new facets, and iv) to facilitate the portability of applications towards new technologies or platforms.

In order to describe our approach, we discuss the SMARTTOOLS features and provide an example of application which is implemented with SMARTTOOLS.

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. To cope with these

changes, applications need to be more open, adaptable and evolutive. These new constraints in software development have emerged mainly because of the following reasons:

- Firstly due to the increase use of the Internet, applications can no longer operate independently but rather they should be distributed. Data communication between applications and users must thus be taken into account during the whole application life-cycle. One important requirement is to choose an adaptative data exchange format.
- The second reason is the proliferation of new component technologies. This increases the difficulty in choosing which component technology will be the most adaptive and evolutive, according to the context of use. For instance it will be necessary to decide whether it is more appropriate to use CCM (*CORBA Component Model*), EJB (*Enterprise Java Bean*), or COM (*Component Object Model*).
- The third reason is the democratization (widespread) of computer science. Users have different knowledge, different needs, a wide range of visualization devices, and specific activity domains. This aspect should be considered when designing and developing applications.
- The last reason is business related. To be more competitive, a company must be able to quickly and cheaply adapt its software in order to meet new user needs and technology evolution.

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. In order to better address these new challenges, we propose an approach which relies on the MDA (*Model-Driven Architecture*) approach, Component Programming, and Generative Programming [3]. It promotes the following key-ideas:

- When a software is being designed and implemented, different concerns are addressed by the programmer. These concerns are better handled if a dedicated model¹ exists for each of them. Among the possible concerns, there are the design of the application data-model, the persistence management, the security specification, the GUI (*Graphical User Interface* definition, and the handling of software components.
- If each model (dedicated to one of the concerns) is independent from technology, then it is possible to capture the know-hows of an application independently from the context of use. Therefore the domain-specific knowledge is much "more reusable".
- When building an application from these models, Generative Programming should be used to glue (assemble) them together according to the context of use (e.g. the technologies). This powerful paradigm enables applications to evolve.

¹ By construction, it will exactly fit to the needs of the concern.

The key-ideas mentioned above influence a lot the implementation of our approach which is compliant with Domain-Driven Development (DDD). The implementation is based on the concept of a software factory [3] and is adapted to the design and implementation of applications which rely on a data model. It provides the ability to define business models and to also perform transformations on them in order to generate either refinements or platform-specific models. In order to achieve this, it uses source code generation as a transformation mechanism in order to produce one or many Platform-Specific Models (PSM), from the Platform-Independent Model (PIM) such as those mentioned above. From our point of view, to rely on several PIMs is particularly relevant when developing open and adaptable applications.

In order to validate our approach, we have developed a software factory, named SMARTTOOLS ² [12], based on this new way of programming. This research prototype has a main goal which is to help the user to produce application tools for different domains. The design of both prototype and applications generated by it, addresses four concerns: the application data model, the writing of semantic analyses, its architecture, and view of the data-model (see figure 1). To each of those concerns, we have associated a model:

- The *data model*. It describes the application structure and should have an application-independent format in order to cut from the technology-specific details.
- The *semantics of both the data model and the application*. It integrates several facilities in order to structure and to modularize the code. This should improve the maintainance of the code and enable easier code reuse.
- The *view model*. Several views of a data model can be defined, such as a structured editor in order to more easily create and update instances of this model (data). This view model must be device-independent.
- The *component model*. It is as tightly integrated as possible with the application requirements. In particular, it enables to specify the provided and required services.

The generators associated with those models will handle the generation of the application, providing the glue to make it works on a specific platform, according to the context of use. If the platform or the underlying technology evolves, it is not necessary to update the models which represent the domain-specific know-hows, but the generators only. The experience gained through developing SMARTTOOLS *i*) provides a more precise description of the approach and, *ii*) demonstrates how the approach favors the possible adaptations of an application according to the future evolutions of the software platform.

This paper is divided in five parts, plus one part dedicated to the related work and one to the conclusion and perspectives. The four first sections describe the main models provided by SMARTTOOLS (data, semantics, view, and component models). For each model (from section 2 to 5), we present the main aspects of the model and we insist on the impact of using both MDA approach and generative

² <http://www-sop.inria.fr/smartool/SmartTools/>

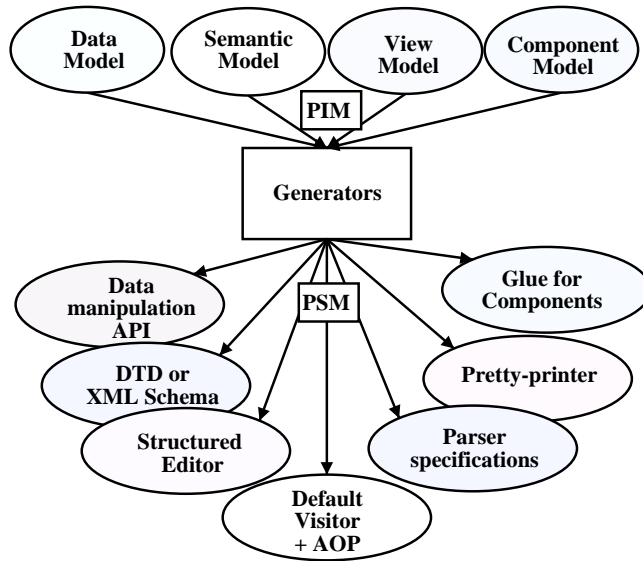


Fig. 1. MDA approach in SMARTTOOLS

programming and on the interest of using standards. Section 6 gives a concrete example of how SMARTTOOLS can be used to develop tools. In this section we particularly focus on the contribution of component-oriented programming.

2 Data Model Generator

For some years, the standardization efforts of both the OMG (*Object Management Group*) and the W3C (*World Wide Web Consortium*) have played major roles in the data and model integration issues. The standard formalisms continuously evolve in order to better address the new needs of applications. For instance, to improve document data validation, the DTD (*Document Type Definition*) language has been replaced by more complex and richer data type document meta-languages such as XML Schema or RDFS (*Resource Description Framework Schema*). Another example deals with object-oriented modeling: the UML (*Unified Modeling Language*) approach has evolved toward a domain-specific model definition based on the MOF (*Meta-Object Facility*) meta-formalism [5].

Instead of using the formalisms mentioned above, we have preferred to define our own abstract data meta-model which *i*) enables associating the semantics using separation of concerns, *ii*) is suitable for the description of small business models and *iii*) is independent from any formalism. This meta-model aims to define the business models associated with an application; it is called ABSYNT. It is simple and close to Abstract Syntax Tree (AST) definitions as shown in the

left part of figure 2. This figure describes the data model of our GUI application (see section 5.3) named Layout Markup Language (LML). It is composed of type and operator definitions, and attribute declarations (an attribute is a piece of information attached to either a type or an operator). For example, *FS* represents a type which may have two implementations: either the *frame* operator or the *set* operator that have both the attribute *title*. The right part of the figure provides an equivalent DTD definition in order to enable the reader to better understand the meaning of our meta-model.

<pre> Formalism of lml is Root is Layout; Operator and type definitions { Layout = layout (FS[] fs); FS = %Frame, %Set; Frame = frame (Set[] set); Set = set (VGroup view); VGroup = split (VGroup view1, VGroup view2), view (); } Attribute definitions { REQUIRED title as Java.Lang.String in frame, set, view; REQUIRED orientation as Java.Lang.String in split; REQUIRED position as Java.Lang.String in split; REQUIRED styleSheet as Java.Lang.String in view; REQUIRED viewType as Java.Lang.String in view; REQUIRED behaviour as Java.Lang.String in view; REQUIRED docRef as Java.Lang.String in view; } </pre>	<pre> <!ENTITY % Frame 'frame'> <!ENTITY % Layout 'layout'> <!ENTITY % Set 'set'> <!ENTITY % VGroup 'split view'> <!ENTITY % FS '%Frame; %Set;'> <!ELEMENT layout (%FS;*)> <!ELEMENT frame (%Set;*)> <!ATTLIST frame title CDATA #REQUIRED> <!ELEMENT set (%VGroup;)> <!ATTLIST set title CDATA #REQUIRED> <!ELEMENT split (%VGroup;,%VGroup;)> <!ATTLIST split position CDATA #REQUIRED orientation CDATA #REQUIRED> <!ELEMENT view EMPTY> <!ATTLIST view viewType CDATA #REQUIRED styleSheet CDATA #REQUIRED behaviour CDATA #REQUIRED docRef CDATA #REQUIRED title CDATA #REQUIRED> </pre>
--	---

Fig. 2. Equivalent data model definitions of LML: an ABSYNT definition on the left, a DTD on the right

This meta-model can be used to define the abstract syntax of existing programming languages as well as Domain-Specific Languages (DSL); it is the cornerstone for all the generated tools and components specified within SMARTTOOLS. The goals of using such a meta-model are the followings:

- To cut off from existing formalisms; For example to open SMARTTOOLS towards applications based on XML (*Extensible Markup Language*) or UML standards;
- To benefit from the development efforts (tools) made around these standards.

Impact of the MDA approach and Generative Programming

The openness of a data-model to standards is as much important as its expressiveness. In order to ensure that, we rely on generators and on model transformations. For instance, we have defined translators (in both ways) between our

meta-model and the DTD or the XML Schema meta-models. Thanks to these translators, it is possible to accept either a DTD, an XML Schema, or an ABSYNT document to describe a data model in SMARTTOOLS. SMARTTOOLS also accepts UML as a data model definition (see the left part of figure 4 - HUM notation). From this data model representation (PIM), it is possible to generate, as shown in figure 3, the following capabilities:

- *An API*. This API provides help for the manipulation of abstract syntax trees (for instance, in order to write semantic analyses);
- *An equivalent DTD or XML Schema*. With this capability, designers can easily export their data models;
- *An editor guided by the syntax*. It is a basic view that may be generated automatically in order to facilitate the handling of data (instances of the model).

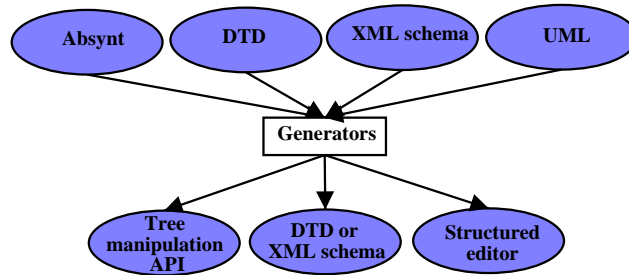


Fig. 3. Generated tools from the data meta-models

2.1 Reuse of Existing Technologies

The openness to standards has an interesting side-effect: it enables the use of APIs related to the standards. For example, in order to avoid the design and the implementation of another proprietorial tree manipulation API, we have chosen the DOM (*Document Object Model*) API standard as the tree kernel. In this way, the code dedicated to tree manipulations which is specific to SMARTTOOLS is minimal and therefore easy to maintain. Moreover our tree implementation benefits from any new service and bug fixes when this standard and its different implementations evolve. Therefore our tree implementation is open, evolutive and can benefit from any DOM-compliant tool or service. For example, all the trees manipulated in SMARTTOOLS can be serialized in XML (see figure 5), transformed with XSLT (*Extensible Stylesheet Language Transformation*), or addressed with XPath for free as these services are provided by the DOM API.

<pre> layout : { fs : FS [*]; }; FS : { title : String; }; frame : FS { set : Set [*]; }; set : FS { view : VGroup ; }; VGroup : { }; split : VGroup { orientation : String; position : String; view1 : VGroup [1..1]; view2 : VGroup [1..1]; }; view : VGroup { title : String; styleSheet :String; viewType :String; behaviour :String; docRef:String; }; </pre>	<pre> <xsd:element name="layout" type="layoutType"/> <xsd:complexType name="layoutType"> <xsd:complexContent> <xsd:sequence> <xsd:group ref="FSType" minOccurs="0" maxOccurs="unbound"/> </xsd:sequence> </xsd:complexContent> </xsd:complexType> <xsd:element name="view" type="viewType"/> <xsd:complexType name="viewType"> <xsd:complexContent> <xsd:extension base="xsd:VGroupType"> <xsd:attribute name="styleSheet" type="xsd:String" use="required"/> <xsd:attribute name="title" type="xsd:String" use="required"/> <xsd:attribute name="viewType" type="xsd:String" use="required"/> <xsd:attribute name="behaviour" type="xsd:String" use="required"/> <xsd:attribute name="docRef" type="xsd:String" use="required"/> </xsd:extension> </xsd:complexContent> </xsd:complexType> </pre>
--	---

Fig. 4. Equivalent data model definitions of LML: on the left UML, on the right an XML Schema

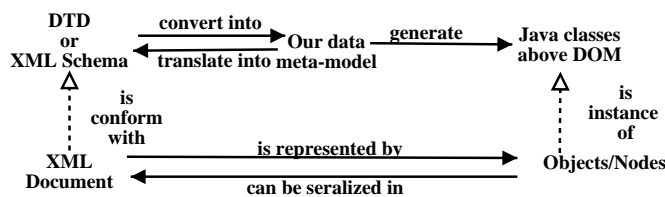


Fig. 5. Bridge between data meta-models and documents

3 Semantics Models

New programming paradigms such as AOP [10], SOP [6] or Generative Programming [3], appear in the last ten years. In a certain way, the "Gang of Four" (GOF) book [4] was already dealing with the problems associated with designing more modular, flexible and extensible software through the proposal of design patterns. One of them is the *visitor* design pattern; it separates the data structures (a hierarchy of classes) and the associated treatments. These treatments are written in a modular way (one class), making easier any modification or extension.

In SMARTTOOLS, we aim to allow the developer to semantically analyse the data, for example to check its validity (type checker), to retrieve some pieces of information, or to evaluate it (interpreter). Such analyses may have a specific tree traversal strategy and use some variables for computational purposes. The

language designers (who may not have a deep knowledge in computer science) should only focus on the information to query within the model, not on the technical issues. Additionally, these queries should be easy to modify and to extend, even at runtime.

To meet these requirements, we have chosen to implement the *visitor* design pattern according to the needs which are commonly required for the analysis (traversal strategy and *visit* method signatures). Based on these needs, we have defined the semantics model. Both semantics and data models are used by our generator to produce a default visitor which visits only the nodes included into the traversal strategy. To write a new semantic analysis involves extending through inheritance the default visitor and overriding some of its *visit* methods in order to specify the suitable treatment.

The capability to be able to extend a semantic analysis dynamically (at runtime) is possible due to a dynamic AOP technique dedicated to our semantics model. For its implementation, instead of using static source code transformations or reflexive mechanisms, we choose to produce the code to be inserted. The integration of this capability into our semantic analysis is performed through an extension of the visitor generator which must produce also the specific aspect-plugging code. It is embedded into the visitor defined by default. Thanks to this extension, the semantic analysis attached to one data model can be extended, not only by inheritance, but also (dynamically) with aspects. The main advantage of such an approach is to provide AOP facilities which are *i)* close to the needs of the model designer, *ii)* easy to use: the resulting description of the operational semantics is simple to understand, and *iii)* straightforward to implement so that it may quickly integrate new needs or potential evolutions.

To summarize, the handling of the semantics of business models is split into three distinct parts: the data model representation, the management of its recursive traversal, and the semantics actions (the behavior) to be attached. Our choice to introduce a separation of concerns even at the implementation level favors easy model transformations and has an interesting impact on the ability of SMARTTOOLS to support applications based on the MDA approach.

4 Component Model Generator

Many component technologies have been proposed such as COM and DCOM by Microsoft, CCM by the OMG, and EJB by 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*). According to the state of the art [14], three of the main challenges in component technologies are the followings:

- *To extend the classical method-call.* In this way, the runtime environment (in a three-tiers architecture, the Internet, a message service, or a database access) can be taken into account without any modification to the business code ;

- *To extend the notion of interface.* The provided and required services can be described and discovered (for example, with the introspection in Java Beans), and the interface can dynamically be adapted (for example, the multi-interface notion in CORBA).
- *To add meta-information to a component.* This is a generic approach to record information dealing with several concerns such as the deployment management or the security policy.

As SMARTTOOLS generates and imports component, it was vital to have a component architecture for its evolution and to make easier the interconnections with tools. Having a component architecture for a factory tool is also useful to be able to build an application with only the required components.

4.1 Abstract Component Model

Instead of using any existing component technology, we have preferred to define an abstract component model i.e. independent from any component technology to clearly express the needs of SMARTTOOLS. Without this model, these needs would have been hidden by the use of a component format (for example, IDL - *Interface Definition Language*) which is not dedicated to our application.

When building the component model it is necessary to take into account the aim of SMARTTOOLS which is to define new data model, to query it and to import existing model representations. One of the consequence of this is that very often components are related to one data-model even if this is not mandatory. This is why the SMARTTOOLS component model is strongly linked with the meta-model which describes data models. This means that the components may be built knowing the data-model representation. This will influence the way components may interact one with the others.

From a component model instance, a generator can automatically produce the non-functional code, that is to say the container that hides all the communication and interconnection mechanisms. For example, the broadcast mechanism used to propagate any modification made on a logical document to its associated views (see section 5) is totally transparent for the designer of an application.

Figure 6 gives an example of the textual description (in XML) of a graphical component whereas figure 7 shows its associated visual representation and especially its connectors.

4.2 Reuse of Existing Technologies

We explain in section 5 that components may interact one with the others, according to the purpose of their data models; this means to be able to exchange data. Due to the use of DOM (see section 2.1), we may benefit from all XML facilities. For instance, we have access to the serialized form of the documents. In particular, it allows components to exchange complex information such as subtrees or path information using XPath. One of the main advantages is that all the components which conform with the same data model can exchange complex pieces of information between their business code.

```

<component name="graph" type="graph" extends="abstractContainer">
  <containerclass name="GraphContainer"/>
  <facadeinterface name="GraphFacade"/>
  <dependance name="koala-graphics" jar="koala-graphics.jar"/>
  <attribute name="nodeType" javatype="java.lang.String"/>
  <input name="addNode" 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>

```

Fig. 6. Component model: XML representation

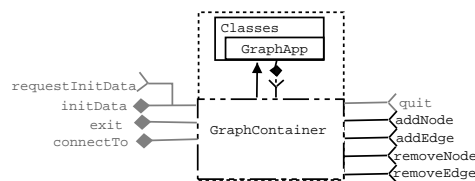


Fig. 7. Component model: graphical representation

4.3 Flexibility of Component Configuration

Indeed, our connection process is much more flexible and dynamic than those offered by the technologies mentioned earlier and which are 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.

Moreover, our component and deployment models (see an example in figure 9) are described in XML format. Our component manager uses these two neutral (XML) formats to instantiate components and to establish connections between them. Figure 8 summarizes the operations performed by our component manager and also the various XML files that are used.

4.4 Impact of MDA Approach and Generative Programming

We showed above that there are many advantages to create an abstract component model which fits with the SMARTTOOLS requirements rather than using a non-specific model. With the integration of an MDA approach (based on generative programming), we are able to produce from our abstract component model the implementations (Platform Specific Models) towards well-known component technologies such as Web-Services, CCM, or EJB (see figure 10). The experience gained by building those projections, make us believe that none of the component technologies mentioned above (Web-Services, CCM, EJB) would have fitted

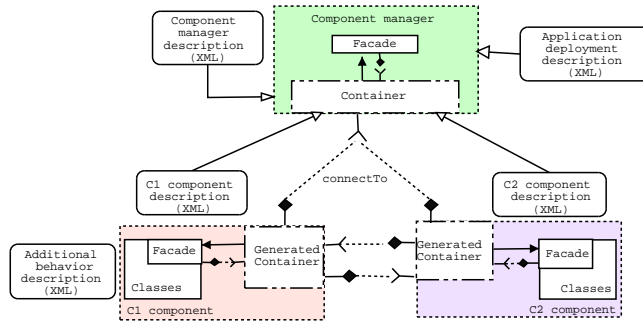


Fig. 8. Functional diagram of the component manager

```

<?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"/>
  <connectTo id_src="ComponentManager" type_dest="glayout">
    <attribute name="docRef" value="file:resources/lml/demo.lml"/>
    <attribute name="xslTransform" value="file:resources/xsl/lml2bml.xsl"/>
    <attribute name="behaviors" value="file:resources/behaviors/bootbehav.xml"/>
  </connectTo>
</application>

```

Fig. 9. Deployment file of the application show in figure 13

with our needs according to the component model itself and to the specifications of connections between models. From our point of view, they are suitable for distributed applications but not for applications with a generic (thus configurable) GUI.

With such an approach, the exportation of the produced components is easier and our models as well as the generated ones can evolve and be much better adapted. Moreover, the architecture of the produced applications is *i*) minimal (only the essential components may be deployed), *ii*) much more flexible, and *iii*) dynamic as new components can very quickly and at any time be plugged in.

5 View Model Generator

The graphical interfaces that make applications interactive must be able to evolve itself according to the application changes. Two main challenges, when designing a graphical interface, should be kept in mind: the interface might be executed on different visualization devices and also through a Web interface. Moreover, the proliferation of new business models requires the ability to quickly design and implement interfaces (or pretty-printers) which are specific to one model or domain. In this context, visual programming can be very useful to build programming environments dedicated to non-complex business models.

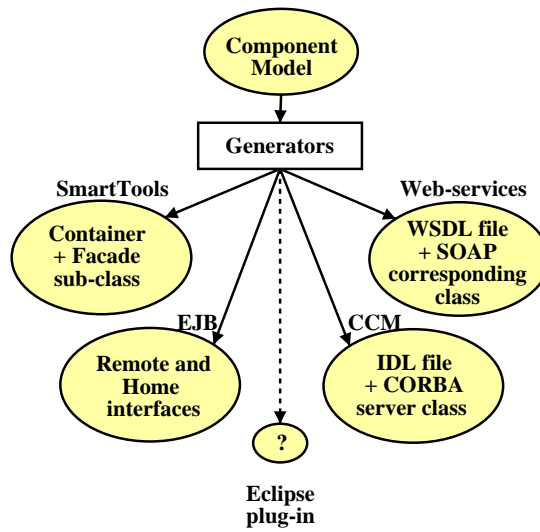


Fig. 10. Component model transformations

One of the goals of SMARTTOOLS is to provide facilities for the development of new tools or programming environments, especially for non-complex description languages. Its design takes into account the specificities of these languages: *i)* they have their own data description language that should be accepted as input, and *ii)* the designers of such languages may not have a deep knowledge in computer science.

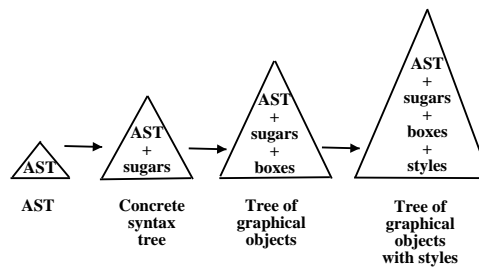


Fig. 11. Graphical view obtained by successive model transformations

For this purpose we define a specific language, named COSYNT, which allows the user to define its user-syntax own domain-specific language. This will be possible due to the capabilities provided by COSYNT for the handling of tree transformations on the logical structure (data-model). These different outputs

are obtained through a sequence of model transformations or refinements (see figure 11). In figure 12 we show an example of this COSYNT specification. The first transformation is described in *BNF* (*Backus Normal Form*) part, the second one in the *Transformation Rules* part and the third one is defines with a CSS (*Cascading Style Sheet*) file.

```

Cosynt for lml is
Concrete Syntaxe {
BNF {
  layout (fslist) : "layout:" *[] #fslist ] ;
  frame (setlist) @title : "frame title: " @title *[] #setlist ] ;
  set (vg) @title : "set title: " @title #vg ;
  split (left, righth) @orientation,@position :
    "split Orientation =" @orientation "Position =" @position
    #left #righth ;
  view () @title,@styleSheet,@docRef,@behavior,@viewType, @transform:
    "[view] title:" @title
    " Document =" @docRef " Transformation =" @transform
    " StyleSheet =" @styleSheet " ViewType =" @viewType
    " Behavior =" @behavior ; }
Parser[k = 3] { }
Lexer[k = 1, attributes = VAR] {
  VAR = <('a'..'z'|'A'..'Z'|'_'|'$')('a'..'z'|'A'..'Z'|'_'|'0'..'9'|'$')*> ;
  INT = <('0'|((('1'..'9')('0'..'9')*))> ; }
}
Layout { Styles[default = null, sugars = keyword, attributes = null] { }
Transformation Rules {
  layout : line: #1 childbox:#2 ;
  frame : line : (#1 #2) childbox:#3 ;
  set : line : (#1 #2) childbox:#3 ;
  split : line : (#1 #2 #3 #4) #5 #6 ;
  view : line : (#1 #2) childbox:(#3 line:#4 #5 line:#6 #7 line:#8 #9 line:#10 #11 line:#12);}
Output {
  BML[default=fr.smarttools.core.view.Sbox, sugars=fr.smarttools.core.view.Slabel,
  attributes=fr.smarttools.core.view.Alabel] { }
  Text[default=sameline, sugars = null, attributes = null] { } }
}

```

Fig. 12. Example of COSYNT file which was use in application show in figure 13

5.1 Reuse of Existing Technologies

More precisely, due to the tree abstract transformations (independent from technologies) described with COSYNT, the COSYNT generator produces i) a ANTLR parser and a XSL Transformation: one from the user-defined syntax to the data-model reification and another one which do the reverse operation in order to be able to save, for example, modifications made through a structured editor and, ii) an XSL Transformation which produces a graphical view (based on Java Beans) of the data-model reification.

5.2 Impact of MDA Approach and Generative Programming

This COSYNT generator is a typical example of a MDA component. It takes as input a data-model and the description of the transformation to be made on

it, using a dedicated transformation language and it produces (outputs) various implementations (XSLT and CSS files, user-defined language parser) of these transformations. To provide such components is particularly suitable for software development because i) it allows the user to define a domain-specific language which is independent from a particular technology and dedicated to the data-model, and ii) to produce automatically symmetrical and incremental transformations which are based on standards.

5.3 GUI Implementation using this Approach

This approach by transformations used for the specification of graphical views is applied to the description of the application GUI. Indeed, a GUI can be considered as a tree of graphical objects (windows, tabs, panels, views, menus, etc.). By using the same approach on this data-model, we can reuse all the tree manipulation methods (insert a node, remove a node, etc.) and the features provided by the view model described above: the GUI is only a particular view of this tree and can be serialized. For example, figure 13 shows two examples of the GUI specification: one in XML on the left, and one using a specific syntax on the right. It is interesting to note that these two GUI representations correspond to the GUI which is displayed within the figure itself. To summarize, we may say that the description of a GUI is encapsulated in one SMARTTOOLS component whose models (data-model, semantics model, component model and view-model) will be addressed in section 6. This section takes the example of the component GUI to explain how to develop an application with SMARTTOOLS.

6 How to develop an Application with SmartTools

In this section, we explain step by step how to build an application with SMARTTOOLS. Our objective is to enable the reader to understand the overlap and the relationships between the four models. In order to achieve this objective, we take an example of application which is the SMARTTOOLS GUI (see figure 13) which is described with the LML model. For this application, we provide all the configuration files and models that are required.

How to Launch an Application In order to launch this application, SMARTTOOLS uses a deployment file (see figure 9). In particular it is necessary to specify the components which are used (mainly the LML component and the view component³) by this application. This file will be used to create a communication channel between the component manager and the view component which requires the use of a configuration file (*file:resources/lml/demo.lml*). For this application, the figure 14 shows the created components and their interconnections.

³ Both of them are SMARTTOOLS components.

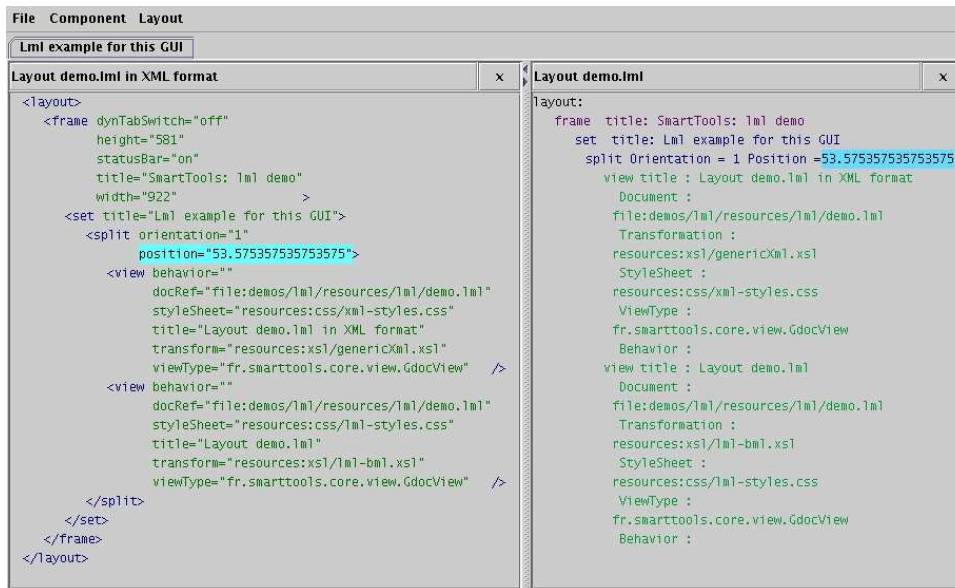


Fig. 13. Two views of the GUI description: a XML view on the left, a textual view on the right.

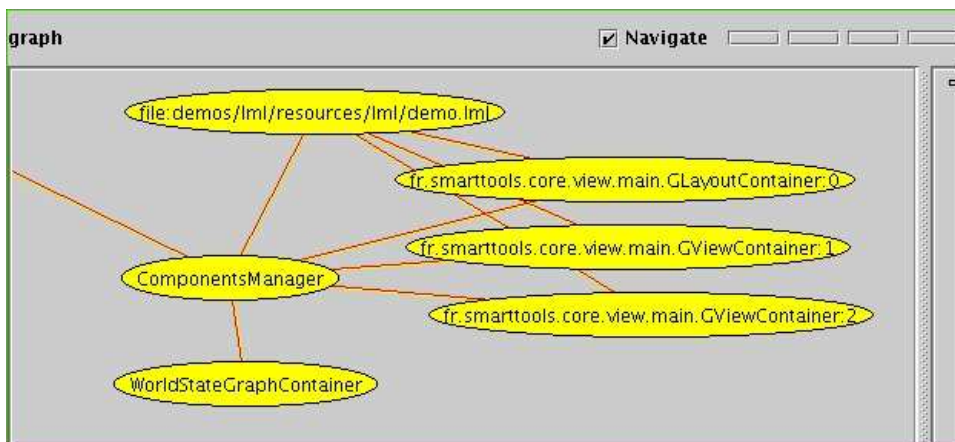


Fig. 14. Open Components and inter-connection of the application shown in figure 13

Models of this Application To describe the logical LML component, it is necessary to define the models described in the previous sections.

- An instance of the component model (see figure 15) which specifies the services;

- The data-model (see figure 2) which is the logical representation of the LML language;
- A view model which represents a view of LML (see figure 12) and which enables to show a syntactical representation of a LML document as it is displayed in the left part of figure 13.

```

<component name="lml" type="document" extends="logicaldocument" >
  <formalism name="lml" file="lml.absynt" dtd="lml.dtd"/>
  <containerclass name="LmlContainer"/>
  <facadeclasse name="LmlFacadeFacade" userclassname="LmlFacade"/>
  <parser type="xml" classname="lml.parsers.LmlXMLParser"> <extention name="lml"/>
  </parser>
  <lml name="DEFAULT" file="resources:lml/lml-default/lml"/>
  <behavior file="resources:behaviors/lml-behaviors.xml"/>
  <input doc="update tree" method="update" name="update">
    <attribute doc="transformation to apply" javatype="java.lang.String"
      name="transformationName"/>
    <attribute doc="orientation" javatype="java.lang.String" name="orientation"/>
  </input>
</component>

```

Fig. 15. Description of LML component

Different Layers from a Component Due to the different generators, the various layers which compose the source code of a component are presented in the figure 16.

- CDML generator produces the container layer ;
- COSYNT generator produces XLST, CSS and parser layers ;
- ABSYNT generator produces the logical layer ;

The set of Component in SmartTools The components of SMARTTOOLS are the components related to the implementation of the various models and those dedicated to the applications described by SMARTTOOLS. figure 17 shows the current state of the SMARTTOOLS distribution.

7 Related Work

Both our approach and SMARTTOOLS are on the edge of different software engineering domains and many related research works. For those reasons we have preferred drawing up the main advantages of the approach instead of trying to compare both of them directly with their respective related works. We show the advantages of this approach according to the openness and evolutivity of the produced applications more than to the skills of SMARTTOOLS itself. There is no doubt that on each concern of SMARTTOOLS, the proposed techniques or solutions are certainly less powerful compared to equivalent research works or tools.

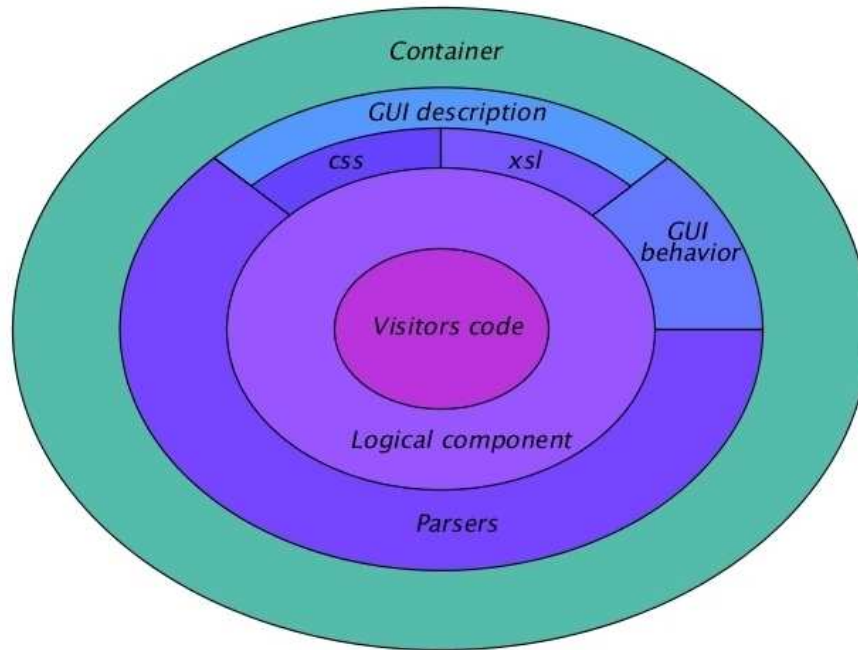


Fig. 16. The various produced layers of a component with SMARTTOOLS

For example our AOP approach is very specific to the models that are addressed by SMARTTOOLS and cannot be compared directly with general approaches or tools such as AspectJ [13].

It is necessary to keep in mind that the core of our approach is to apply to different levels an MDA approach using generative programming. The main benefits of this approach are the followings:

- To handle different concerns homogeneously and simultaneously. On the contrary, the component technologies mentioned earlier are mainly interested in the distribution concern.
- To remain on the implementation level. The UML modeling approaches [7, 8] suffers from the gap between the specification and implementation levels.
- To produce generator-free source code, very close to hand-written programs. Very often, tools such as Centaur [1], FNC-2 [9], or others [11] introduce a strong dependence between the generator and the produced code.
- To be evolutive and open due to the use of standard technologies (e.g. XSLT for program transformation). For example according to program transformation, there are many other tools available [11] but with proprietary input formats and interpreter engines that require additional work to plug in and use them.

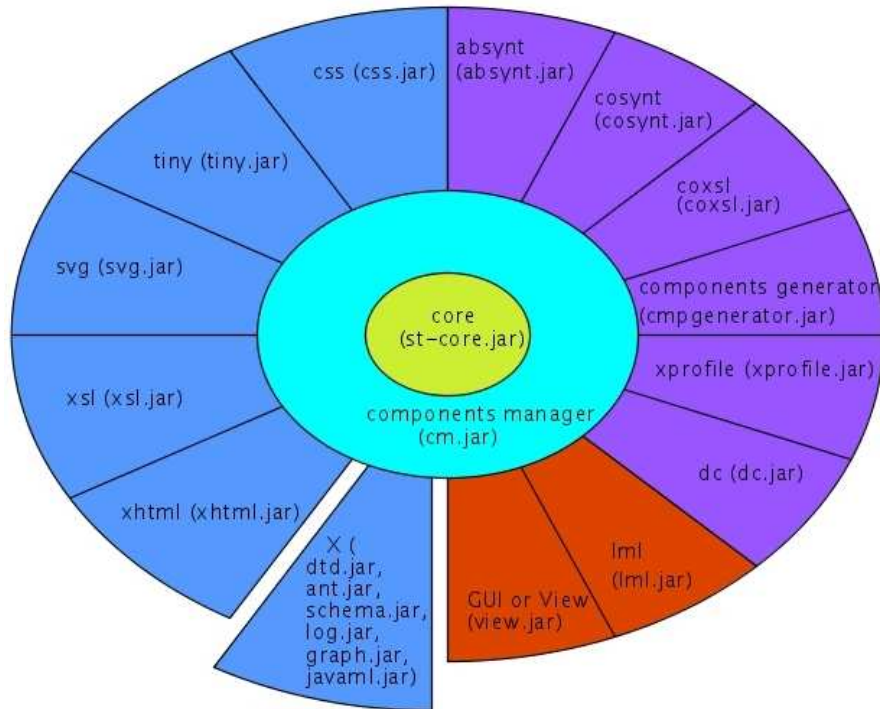


Fig. 17. The various components (jar file) of SMARTTOOLS

- To treat the GUI or other environment facilities as separated entities (components) that may or not, be integrated in the resulting application. This feature does not usually exist in the IDEs (*Integrated Development Environments*) that force the produced applications to be integrated into the IDE framework itself.

8 Conclusion and Perspectives

Through the continuous development of SMARTTOOLS, we are validating a new approach in software development mainly based on transformation and generation of models. We promote the idea that each concern of a model should be described by business models in order to better fit to the requirements. Moreover these models should be independent from the context of use, that is to say from existing technologies. The main benefit is that these technologies should be able to evolve independently from the business model and vice-versa. New models based on new paradigms and new technologies are built due to generative programming. They represent either a refinement of the input model (another PIM) or its implementation on a dedicated software platform (a PSM).

The main advantage of this approach is to make the evolution of models easy according to the software platform evolution or to the creation of new concerns. This evolution is performed only through modifications of the generators associated with each model (data model, component model, view or GUI model, etc.). These generators contain the design methodologies (they represent altogether the software tool factory). They are customized due to input models, and they produce new intermediate models (which may represent refinements) or the final models adapted to the software platform.

In the short-term, we wish to propose new facilities for the data meta-model specification and to allow the description of at least one part of the semantics by a declarative approach. Regarding the data meta-model we aim to propose a new one (called SmartModels), which is closer to the MOF and which introduces three additional capabilities: *i*) an entity may have a meta-level, *ii*) an entity may be generic (this is particularly useful to describe lines of products) and *iii*) an entity may be specialized. Regarding the semantics model, we propose a clear separation between the semantics of business models and the semantics of application. Roughly, one may say that the approach provided for the description of the application semantics will strongly rely on the approach described in section 3, but will offer a specific entity (the *facet*) which will encapsulate this semantics. A facet is close from the subjects (Subject-Oriented Programming) and will contain information on the associated application in order to improve the generation capabilities.

The description of the semantics of a business model must be as much declarative as possible and most of all must be independent from the software platform in order to capitalize the business knowledge and to protect it against technological evolutions. We introduce this semantics mainly within the entity meta-level through *i*) assertions which rely in particular on OCL, *ii*) a set of parameters and properties which are used especially for the description of generic entities, and *iii*) *actions* which describe the behavior of entities (they strongly depend on the parameters and properties⁴ [2]).

According to SMARTTOOLS approach, the implementation of the work described above is fully consistent with it. The improvements dedicated to the description of both reification and semantics as it is mentioned above are made using the description and generation capabilities provided by SMARTTOOLS and described within the previous sections. SmartModels will be provided as an alternative to the formalisms already included in SMARTTOOLS and the user may choose according to its needs and to the community they belongs to, one model instead of another.

⁴ We are thinking about different solutions such as defining a dedicated pseudo-language, but we are looking also to other related works such as UML and Action Semantics.

References

1. Patrick Borras, Dominique Clément, Thierry Despeyroux, Janet Incerpi, Gilles Kahn, Bernard Lang, and Valérie Pascual. CEN TAUR: the System. *SIGSOFT Software Eng. Notes*, 13(5):14–24, November 1988.
2. A. Capouillez, P. Crescenzo, and P. Lahire. OFL: Hyper-Genericity for Meta-Programming - An Application to Java. Technical Report I3S/RR–2002-16–FR, Laboratoire d’Informatique, Signaux et Systèmes de Sophia-Antipolis, avril 2002. <http://www.crescenzo.nom.fr/>.
3. Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Techniques, and Applications*. Addison-Wesley, June 2000.
4. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, 1995. ISBN 0-201-63361-2-(3).
5. Object Management Group. Meta Object Facility (MOF) specification (version 1.3). Technical report, Object Management Group, March 2000.
6. William Harrison and Harold Ossher. Subject-oriented programming (A critique of pure objects). In Andreas Paepcke, editor, *Proceedings ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 411–428. ACM Press, October 1993.
7. Jean-Marc Jézéquel, Wai-Ming Ho, Alain Le Guennec, and François Pennaneac’h. UMLAUT: an extendible UML transformation framework. In Robert J. Hall and Ernst Tyugu, editors, *Proc. of the 14th IEEE International Conference on Automated Software Engineering, ASE’99*. IEEE, 1999.
8. Jean-Marc Jézéquel, Heinrich Hußmann, and Stephen Cook, editors. *UML 2002 - The Unified Modeling Language, 5th International Conference, Dresden, Germany, September 30 - October 4, 2002, Proceedings*, volume 2460 of *Lecture Notes in Computer Science*. Springer, 2002.
9. Martin Jourdan, Didier Parigot, Catherine Julié, Olivier Durin, and Carole 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).
10. Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet Akşit and Satoshi 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.
11. Paul Klint. A Meta-Environment for Generating Programming Environments. *ACM Transactions on Software Engineering Methodology*, 2(2):176–201, 1993.
12. Didier Parigot, Carine Courbis, Pascal Degenne, Alexandre Fau, Claude Pasquier, Jol Fillon, Christophe Help, and Isabelle 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).
13. Renaud Pawlak, Lionel Seinturier, Laurence Duchien, and Gérard Florin. JAC: A flexible solution for aspect-oriented programming in Java. *Lecture Notes in Computer Science*, 2192:1–24, 2001.
14. Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 1998.