Domain-Driven Development: the SmartTools Software Factory

Didier Parigot
INRIA Sophia-Antipolis
2004, route des Lucioles BP93
F-06902 Sophia-Antipolis cedex - France
Didier.Parigot@inria.fr

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

Abstract

With the increasing dependency on 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 logic. 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 generators associated with domain-specific languages, each of the latter related to one possible concern useful when building an application. It relies on Generative Programming, Component Programming and Aspect-Oriented Programming. A software factory, called SmartTools*, has been developed using this new approach.*

The main results are i) to build software of better quality and to enable rapid development due to Generative Programming and, ii) to facilitate insertion of new facets and the portability of applications to new technologies or platforms due to business logic and technology separation.

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. As a result, applications need to be more open, adaptable and capable of evolving. These new constraints in software development have emerged primarily due to the following reasons:

- Firstly, due to the increase use of the Internet, applications can no longer operate independently but rather they should be distributed. Therefore, data communication between applications and users must be taken into account during the whole application life-cycle. One important requirement is to choose a well-known 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 adaptable and capable of evolving, according to the context of use. For instance, it is 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 feature 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. In order to better address these new challenges of openness, flexibility, and evolution, we propose an approach which relies on the MDE (*Model-Driven Engineering*) approach, Component Programming, and GP (*Generative Programming*) [5]. It promotes the following key-ideas:

- When software is being designed and implemented, different concerns are addressed by the

programmer. These concerns are better handled if a dedicated meta-model[1] exists for each of them.

- If each meta-model (dedicated to one of the concerns) is independent from any technology, then it is possible to capture the expertise of an application independently from the context of use. Therefore, the domain-specific knowledge is much "more reusable".
- When building an application, Generative Programming (GP) should be used to glue (assemble) the models together according to the context of use (e.g. the technologies). This powerful paradigm enables applications to evolve.

In order to validate our approach, we have developed a software factory, named SmartTools [2] [15], based on this new way of programming, which is compliant with the Domain-Driven Development (DDD) approach [4]. The principal goal of this research prototype is to propose a tool which demonstrates that, with new development methods, it is possible to produce more quickly open and adaptable applications compared with the classical development methods. The implementation is based on the concept of a software factory [7] and is adapted to the design and implementation of applications which rely on a data model. It provides the ability to define domain-specific languages (DSLs) and also to perform transformations on them in order to generate either refinements or platform-specific models.

The design of both prototype and applications generated by it addresses five concerns (see Figure 1): the application data model, the writing of semantics analyses, its architecture, the views of the data model, and its graphical user interface. To each of those concerns, we have associated a meta-model[3]:

- The *data meta-model*, named AbSynt. It describes the application structure and should have an application-independent format in order to cut from the technology-specific details. More precisely, the AbSynt language is a meta-language (meta-meta-model) which is used to define languages (meta-model);
- The *semantics meta-model of both the data model and the application*, named ViProfile. It integrates several facilities in order to structure and to modularize the code. This should improve the maintenance of the code and enable easier code reuse;
- The *view meta-model*, named CoSynt. 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 (programs or documents). This view model must be device-independent.
- The *component meta-model*, named Cdml. It is as tightly integrated as possible with the application requirements. In particular, it enables to specify the provided and required services.
- The *GUI meta-model*, named Lml. It describes a possible configuration of a GUI for a given application.

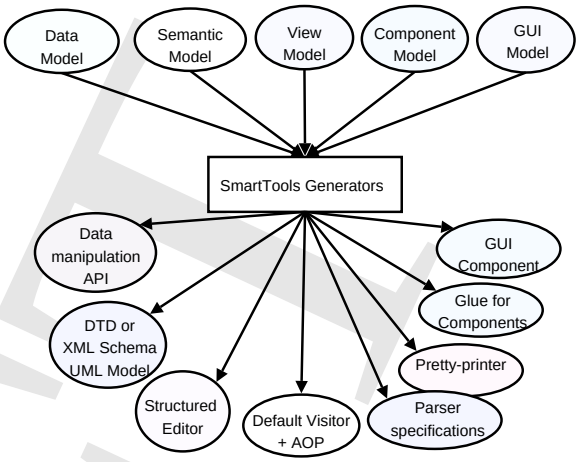

Figure 1. MDE approach in SmartTools

The generators associated with those meta-models handle the generation of the application, providing the glue to enable it to work 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 meta-models which represent the domain-specific expertise, 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.

The principal contribution of our approach is to show that it is more important to propose a rich set of generators, each specialized in one concern, than

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

[2] `http://www-sop.inria.fr/smartool/SmartTools/`

[3] Sometimes one will use the language term in the place of meta-model term when there can be confusions between meta-model and model term

to propose more generic approaches or strongly specialized ones for a given concern. The limitations of our approach are primarily the following: *i)* strong dependences on the Java programming language and the Swing library; *ii)* our graphic tools are well adapted for simple meta-models and the possibilities of our graphical user interface still remain very basic; *iii)* our two meta-models (AbSynt and Cdml) are minimalist (They will be soon extended to respectively add the inheritance and distribution notions).

This paper is divided in five parts, plus one part dedicated to the related work and one to the conclusion, section 2 provides a concrete example of how SmartTools can be used to develop tools. For instance, we show the example of a graphical user interface (GUI) of SmartTools. The four following sections describe the main meta-models provided by SmartTools (data, semantics, component and view meta-models) based on the GUI application described in Section 2. For each meta-model (from Section 3 to 6), we present the main aspects of the model and we lay stress on the benefits of using both MDE approach and GP as well as on the interest of using standards.

2 Short Overview of SmartTools

SmartTools is heavily bootstrapped; that is it internally uses its technology to develop its own languages and components. Through the development of these languages and components, our approach in integrating the mentioned paradigms and technologies has been intensively tested and refined. 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. Since then, SmartTools has been used to produce tools for many diverse languages (about thirty languages) such as SVG, DTD, XML schema, CSS, WSDL, and BPEL. However the most complete application is SmartTools itself which is composed of the generators associated at each language (AbSynt,CoSynt,Cdml,Lml,ViProfile). The SmartTools framework represents approximately 100 000 lines of Java source code before the generation stage and 1 000 000 lines after. This ratio shows the efficiency of this approach and validates this new development approach based on GP.

In this section we present how a language, our GUI language, can be developed using SmartTools. The main reasons of this choice are that the GUI implementation is strongly bootstrapped and its models (data model, component model, etc...) are easy to understand. This GUI example will be used all along of this paper, for each meta-model.

2.1 GUI Language - Lml

The SmartTools GUI is only a particular graphical view (windows, tabs, panels, views, menu, etc.) of a document (based on the Lml language, Layout Markup Language). A GUI model can be considered as a tree of manipulated models (documents or programs) and their associated graphical views. In this way, we can reuse all the tree manipulation methods (insert a node, remove a node, etc.) and the features provided by the view meta-model (See Section 6). For instance, the GUI of Figure 2 can be serialized into an XML file (see Figure 3). This GUI shows two graphical views of it - one in XML on the left, and one using a specific concrete syntax on the right - and is also itself a view of it.

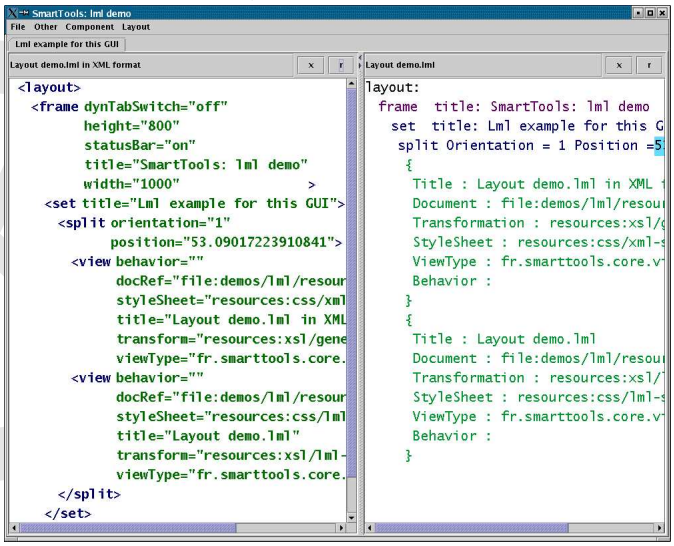

Figure 2. Three views of the GUI model of Figure 3: an XML view on the left, a textual view on the right, and the window itself (interpretation of the model)

2.2 Models of this GUI Application

To describe the Lml component, it is necessary to define the following models:

```
<layout>
 <frame title="SmartTools: lml demo"
        statusBar="on" width="1000"
        height="800" dynTabSwitch="off">
  <set title="Lml example for this GUI">
   <split orientation="1" position="68">
     <view title="Layout demo.lml in XML format"
        behavior=""
        viewType="fr.smarttools.core.view.GdocView"
        docRef="file:demos/lml/resources/lml/demo.lml"
        styleSheet="resources:css/xml-styles.css"
        transform="resources:xsl/genericXml.xsl" />
     <view title="Layout demo.lml"
        behavior=""
        viewType="fr.smarttools.core.view.GdocView"
        docRef="file:demos/lml/resources/lml/demo.lml"
        styleSheet="resources:css/lml-styles.css"
        transform="resources:xsl/lml-bml.xsl" />
    </split>
  </set>
 </frame>
</layout>
```

Figure 3. GUI Model (LML) uses to produce the application shown in Figure 2

- The component model (see Figure 4) which specifies the services of the LML component. With this component model, the classical concepts for a component are described such as the facade, the container, the inputs and outputs, associated with the DSL data model (formalism).
- The data-model (see Figure 5) which define the LML language. This language gives the logical structure of a GUI model: a layout is composed of a set of frames, a frame (window) is composed of a set of sets (tabs), a set is a view or a split, etc.
- A view model which represents easy-to-read views of LML documents such as the one in the right part of Figure 2 or the window/GUI itself. For the left part of this Figure, SMARTTOOLS uses a generic view model (the default one) to produce an XML view, applicable to any model.

2.3 Implementation of the GUI Application

To build this GUI, a particular graphic component was implemented, named `glayout`. This graphic component is particular in the sense that is associated a dedicated graphic object (`layout`, `frame`, `split` and `view`) to each LML entity. A transformation (built on the same principle as for construction of the graphic views, see Section 6) maps the logical entities to the graphic objects to create the GUI. The business logic of this application - the GUI - is only made up of these graphic objects and this transformation.

```
<component name="lml" type="document"
           extends="logicaldocument" >
 <formalism name="lml" file="lml.absynt"
            dtd="lml.dtd"/>
 <containerclass name="LmlContainer"/>
 <facadeclass name="LmlFacadeFacade"
              userclassname="LmlFacade"/>
 <parser type="xml" <extention name="lml"/>
         classname="lml.parsers.LmlXMLParser">
 </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>
```

Figure 4. Component Model (CDML) of LML

```
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; }
```

Figure 5. Data Model (ABSYNT) of LML

2.4 The GUI Application Deployment

In order to launch this application (the GUI), SMARTTOOLS uses a deployment model (see Figure 6). It is necessary to specify the components[4] which are used (mainly the LML component and the `glayout` component) by this application. This deployment model will be used to create a communication channel between the component manager and the `glayout` component which requires the use of a LML document - a GUI model - (*file:resources/lml/demo.lml*, see Figure 3). This model described the initial state of the GUI application, Figure 2. For this GUI application, the Figure 7 shows the created components and their interconnections with this GUI model.

[4] These components are SMARTTOOLS components.

```
<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>
```

Figure 6. Deployment model of the GUI application

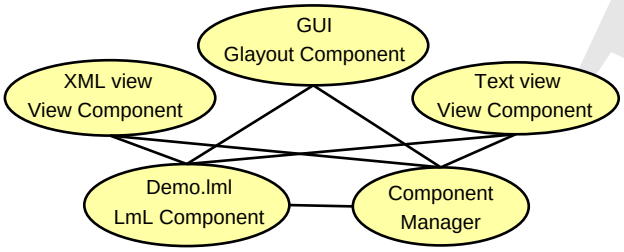

Figure 7. Manipulated Components and interconnections with the GUI application

3 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 resolving 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 [8].

3.1 Data Meta-Model (AbSynt language)

Instead of using the formalisms mentioned above, we have preferred to define our own abstract data meta-model which *i)* enables associating a semantics using the separation of concerns, *ii)* is easy to use to describe non-complex DSLs and *iii)* is independent from any formalism. This meta-model aims to define models associated with an application and is called AbSynt. It is simple and close to Abstract Syntax Tree (AST) definitions as shown in Figure 5. This Figure describes the data model of our GUI application (Lml language, see Section 2). 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* type represents a type which may have two implementations: either the *frame* operator or the *set* operator that have both the attribute *title*. This meta-model can be used to define the abstract syntax of existing programming languages as well as DSLs. It is the cornerstone for all the generated tools and components specified within SmartTools.

3.2 Impact of the MDE and GP Approaches

The openness of a data-model to standards is as 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. Due to these translators, it is possible to accept either a DTD, an XML Schema, or an AbSynt model to describe a data model in SmartTools. SmartTools also accepts UML model (in HUTN notation, UML *Human-Usable Textual Notation*). From this data model representation, it is possible to generate, as shown in Figure 8, the following capabilities:

- *An API.* This API provides help for the manipulation of abstract syntax trees (for instance, in order to write semantics analyses);
- *An equivalent DTD, XML Schema, or UML description.* 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 documents or programs (a model).

3.3 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

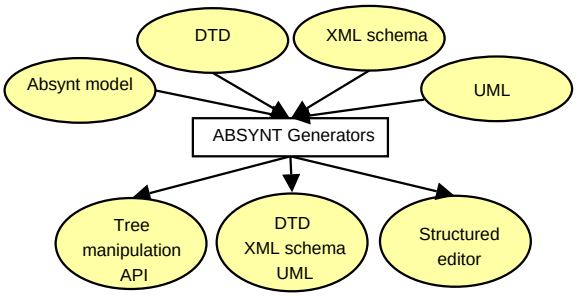

Figure 8. Generated tools from the data meta-models (AbSynt)

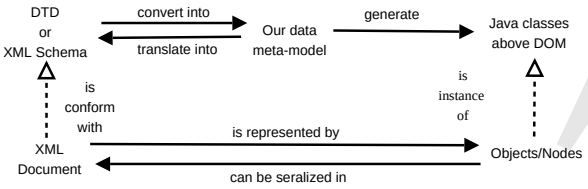

Figure 9. Bridge between data meta-models and models

the implementation of another propriatory 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, 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 (see Figure 9), transformed with XSLT (*Extensible Stylesheet Language Transformation*), or addressed with XPath for free as these services are provided by the DOM API.

4 Semantics Generator

New programming paradigms such as AOP [12], SOP [9] and GP [5] have appeared in the last ten years to provide new ways of developing flexible extensible applications. In a certain way, the "Gang of Four" (GOF) book [6] 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 [14]; it separates the data structures (a hierarchy of classes) from the associated treatments. These treatments are written in a modular way (one class), making easier any modification or extension.

4.1 Semantic Meta-Model (ViProfile language)

In SmartTools, we aim to allow the developer to semantically analyze 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, named ViProfile (Figure 10 gives a semantic model of the Lml language). 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 semantics analysis involves extending through inheritance the default visitor and overriding some of its *visit* methods in order to specify the suitable treatment. The implementation of our generators (in particular the AbSynt and CoSynt generators) strongly use this *visitor* technique, as well as our data meta-model transformations (in Figure 9).

```
XProfile lml;
Formalism lml;
import lml.SymTab;
Profiles
java.lang.Object visit(%Layout, lml.SymTab symTab);
java.lang.Object visit(%FS, lml.SymTab symTab);
java.lang.Object visit(%Frame, lml.SymTab symTab);
java.lang.Object visit(%Set, lml.SymTab symTab);
java.lang.Object visit(%Vgroup, lml.SymTab symTab);
Strategy TOPDOWN;
```

Figure 10. A semantic model (ViProfile) of Lml

The ability to be able to extend a semantics 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 have chosen to generate hooks at the join points. The integration of this capability into our semantics analyses

is performed through an extension of the visitor generator. Due to this extension, the semantics analyses 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 DSL designer, *ii)* easy to us as 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. In [3], we[5] strongly use our AOP approach to develop an extensible and adaptable BPEL engine.

5 Component 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 [17], 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-tier architecture, the Internet, a message service, or a database access) can be taken into account without any modification to the business logic.
- *To extend the notion of interface.* The provided and required services can be described and discovered (for example, with the introspection available within Java Beans), and the interface can dynamically be adapted.
- *To add meta-information to a component.* This is a generic approach to record information dealing with several concerns such as deployment management or security policies.

As SmartTools generates and imports components, it was vital to include a component architecture for its evolution and to simplify the interconnections with external tools. Including a component architecture for a factory tool is also useful to be able to build applications with only the required components.

5.1 Abstract Component Meta-Model (Cdml language)

Instead of using an existing component technology, we decided to define an abstract component meta-model (see Figure 4) i.e. one that is independent from any component technology to clearly express the needs of SmartTools. Without this meta-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 meta-model it was necessary to take into account the aims of SmartTools which are to define a new data model, to query it and to import existing model representations. One of the consequences of this is that very often components are related to one data-model even if this is not mandatory. Therefore, the SmartTools component meta-model is strongly linked with the data meta-model. This means that the components may be built knowing the data-model representation. This influences the way components may interact with each other. From a component model (see Figure 4), 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 6) is totally transparent to the designer of an application.

5.2 Reuse of Existing Technologies

We explain in Section 6 that components may interact with each other by exchanging data. Due to the use of the DOM API (see Section 3.3), all the XML facilities are available. For example, any document can be serialized into an XML form. In particular, it enables components to exchange complex information such as sub-trees 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 logic.

5.3 Flexibility of the Component Configuration

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 (ports). Our component manager uses our component and deployment models (see an example in Figure 6) - two XML formats - to instantiate components and to establish connections between them. Figure 11 summarizes the operations performed by our component

[5] The work of Carine Courbis at UCL.

manager and also the various XML files (models) that are used.

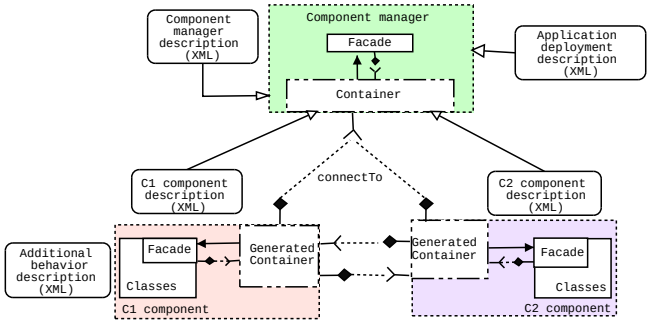

Figure 11. Functional diagram of the component manager

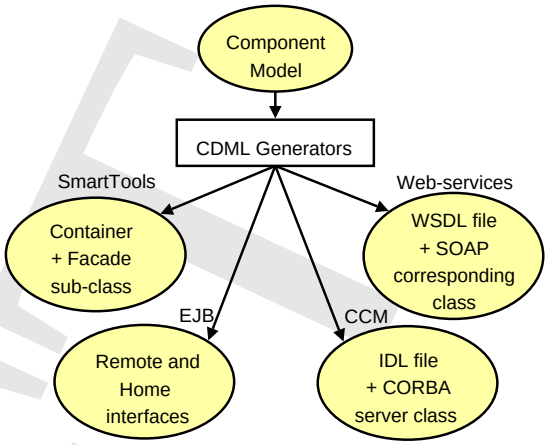

Figure 12. Component model transformations

5.4 Impact of MDA and GP Approaches

As mentionned earlier, there are many advantages in creating an abstract component meta-model which fits with the application requirements rather than using a non-specific model. With the integration of a MDA approach (based on GP), we are able to produce from our abstract component meta-model the implementations (Platform Specific Models) towards well-known component technologies such as Web-Services, CCM, or EJB (see Figure 12). The experience gained by building those projections makes us believe that none of the component technologies mentioned above (Web-Services, CCM, EJB) would have fitted with our needs. 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 DSLs can evolve and be much better adapted. Moreover, the architecture of produced applications is *i)* minimal (only the essential components may be deployed), *ii)* much more flexible, and *iii)* dynamic as new components can be very quickly developed and plugged in at any time.

6 Graphical View Generator

The graphical interfaces that make applications interactive must also be able to evolve themselves 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. The proliferation of new domain-specific 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 when building programming environments dedicated to non-complex domain-specific models.

6.1 View Meta-Model (CoSynt language)

For this purpose, we have defined a specific language, named CoSynt, which enables the designers to define a concrete syntax to their DSL. With CoSynt, tree transformations can be specified based on the data model. These different outputs are obtained through a sequence of model transformations or refinements (see Figure 13).

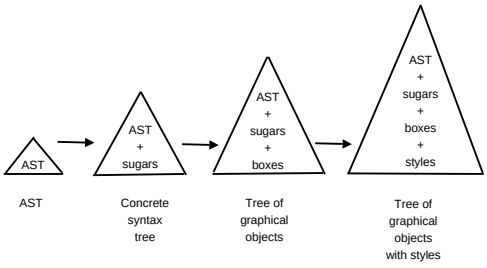

Figure 13. Successive model transformations performed to obtain a graphical view

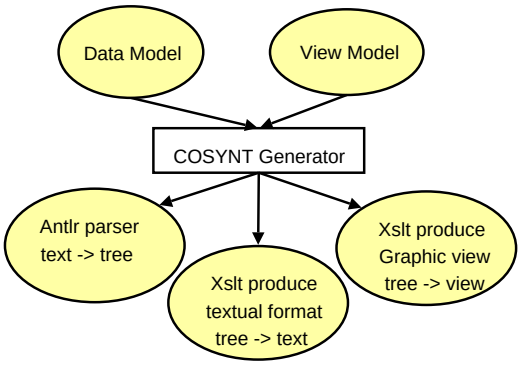

Figure 14. CoSynt **generator**

6.2 Reuse of Existing Technologies

More precisely, due to the tree abstract transformations (independent from any technology) described with CoSynt, the CoSynt generator produces *i)* a ANTLR parser and *ii)*, for the reverse operation - the pretty-printing - two XSLT stylesheets which produce respectively a textual form and graphical view (based on Java Beans) of the document (see Figure 14).

6.3 Impact of MDE and GP Approaches

This CoSynt generator is a typical example of a MDE component. It takes as input a data-model and the description of the transformations to be performed on it, using a dedicated transformation language. It produces (outputs) various implementations (XSLT file, user-defined language parser) of these transformations. To provide such components is particularly suitable for software development because i) it enables the designer to define a DSL which is independent from a particular technology and dedicated to the data-model, and ii) it automatically produces symmetrical and incremental transformations based on standards.

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 work. We have focused on the advantages of this approach according to the openness and ability to evolve of produced applications more than one the skills of SmartTools itself. There is no doubt that on each concern, the proposed techniques or solutions are certainly less powerful compared to equivalent research work or specific tools. For example, our AOP approach is very specific to our semantics analyses and cannot be compared directly with general approaches or tools such as AspectJ [16]. It is necessary to keep in mind that the core of our approach is to apply at different levels an MDE approach using GP. 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 modelling approaches [10] suffer 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 [2, 11, 13], introduce a strong dependence between the generator and the produced code.
- To be capable of evolving and open due to the use of standard technologies (e.g. XSLT for program transformation). For example, there are many other tools available for program transformations [13] but they use proprietary input formats and interpretor engines that require additional effort to plug them in and use them.
- 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 IDEs (*Integrated Development Environments*) [1] that forces produced applications to be integrated into the IDE framework itself.

Finally, our approach is based on the general concepts (abstraction, granularity, specificity) of Software Factory which are described in [7].

8 Conclusion

Through the continuous development of SmartTools, we are validating a new approach in software development mainly based on transformations and generations of models. We promote the idea that each concern should be described by a DSL (meta-model) in order to better fit the application requirements. Moreover, these meta-models should be independent from the context of use, that is from any existing technology.

The main advantage of this approach is that the metamodels are resilient to any evolution in the underlying technologies except when a new concern needs to be added. This evolution is performed only through modifications of the generators associated with each metamodel. 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.

References

[1] Eclipse. `http://www.eclipse.org/`.

[2] 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.

[3] C. Courbis and A. Finkelstein. Towards an Aspect Weaving BPEL Engine. In Y. Coady and D. H. Lorenz, editors, *Proceedings of the Third AOSD Workshop on Aspects, Components, and Patterns for Infrastucture Software (ACP4IS)*, number NU-CCIS-04-04, Lancaster, UK, March 2004. College of Computer and Information Science Northeastern University.

[4] R. Crocker and G. L. S. Jr., editors. *Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2003, October 26-30, 2003, Anaheim, CA, USA*. ACM, 2003. Special track, Krzysztof Czarnecki and John Vlissides.

[5] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Techniques, and Applications*. Addison-Wesley, June 2000. ISBN 0201309777 chapter Aspect-Oriented Decomposition and Composition.

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

[7] J. Greenfield and K. Short. Software factories: assembling applications with patterns, models, frameworks and tools. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 16–27. ACM Press, 2003.

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

[9] 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.

[10] J.-M. Jézéquel, H. Hußmann, and S. 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.

[11] 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.

[12] 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.

[13] P. Klint. A Meta-Environment for Generating Programming Environments. *ACM Transactions on Software Engineering Methodology*, 2(2):176–201, 1993.

[14] 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.

[15] 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).

[16] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: A flexible solution for aspect-oriented programming in Java. *Lecture Notes in Computer Science*, 2192:1–24, 2001.

[17] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 1998.