

LABORATOIRE



INFORMATIQUE, SIGNAUX ET SYSTÈMES
DE SOPHIA ANTIPOLIS
UMR 6070

TO BUILD OPEN AND EVOLUTIVE APPLICATIONS: AN APPROACH BASED ON MDA AND GENERATIVE PROGRAMMING

Carine Courbis, Philippe Lahire, Didier Parigot

Projet OCL

Rapport de recherche
ISRN I3S/RR-2004-21-FR

Septembre 2004

RÉSUMÉ :

MOTS CLÉS :

ABSTRACT:

With the emergence of the Internet and proliferation of new component and distributive technologies, the design and implementation of complex applications has to 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 evolutive. To accomodate 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 (Model-driven Architecture) compliant and relies on Generative Programming. It contributes to the research works on Domain Driven Development (DDD). The main expected results are i) to get softwares of better quality thanks to business models and technology separation, ii) to have a more straightforward code, iii) to allow the rapid development and insertion of new facets and iv) to facilitate the portability of applications towards new technologies or platforms. Finally, in order to illustrate our approach, a generative programming system, called SMARTTOOLS, is presented. It may be compared to a software factory dedicated to applications which rely on one data model.

KEY WORDS :

To Build Open and Evolutive Applications: an Approach based on MDA and Generative Programming *

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

Philippe Lahire
Laboratoire I3S (UNSA/CNRS)
2000 route des lucioles BP 121
F-06903 Sophia-Antipolis CEDEX, France
Philippe.Lahire@unice.fr

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

Abstract

With the emergence of the Internet and proliferation of new component and distributive technologies, the design and implementation of complex applications has to 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 evolutive.

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 (Model-driven Architecture) compliant and relies on Generative Programming. It contributes to the research works on Domain Driven Development (DDD). The main expected results are i) to get softwares of better quality thanks to business models and technology separation, ii) to have a more straightforward code, iii) to allow the rapid development and insertion of new facets and iv) to facilitate the portability of applications towards new technologies or platforms.

Finally, in order to illustrate our approach, a generative programming system, called SMARTTOOLS, is presented. It may be compared to a software factory dedicated to applications which rely on one data model.

*This project is supported in part by the W3C consortium with the IST-2000-28767 QUESTION-HOW Project. We would also like to thank PASCAL DEGENNE and ALEXANDRE FAU for their software development efforts in 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. Before going any further, we explain why these new constraints in software development have emerged.

- The first reason is that the emergence of the Internet requires applications to be no more PC-enclosed but rather distributed. Thus, from now, data communication between applications and users must be taken into account during the whole application life-cycle. One important point is to choose a well-adapted data exchange format.
- The second reason of these changes is the proliferation of new component technologies. It makes difficult to choose the one which is the more adapted and the most evolution-prone according to the context of use. For instance it will be necessary to decide whether it is better to take CCM (*CORBA Component Model*), EJB (*Entreprise Java Bean*), or COM (*Component Object Model*), etc.
- The third reason is the democratization (widespread) of computer science. Users may have now different knowledges, 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. Indeed, to be competitive a company must quickly and cheaply adapt its software to new user needs and technologies.

In software engineering, object-oriented programming is not always sufficient to handle clear designs and reusable developments of software. For example, concerns can be cross-cut between classes and there can be a mix between functional and non-functional code in a single class making the code difficult to maintain and debug. This situation explains the need to provide new programming paradigms such as AOP (*Aspect-Oriented Programming*) [19], SOP (*Subject-Oriented Programming*) [14], IP (*Intentional Programming*) [33], or component programming [35].

At the specification level, a strong and continuous evolution is undergoing towards standards of the W3C (*World Wide Web Consortium*) for documents or of the OMG (*Object Management Group*) for design methodologies such as UML (*Unified Modeling Language*) or the MDA (*Model-Driven Architecture*) approach [13, 37].

It seems that having only one model like in UML to tackle all the possible concerns of applications does not always provide the most comprehensive approach to specify them. Moreover, the semantics associated to each UML entities is not very well specified and is sometimes ambiguous. This is a strong drawback if the model need to be executable.

In order to better address these new challenges, we propose an approach which relies on both MDA and Generative Programming [9]. It promotes following key-ideas:

- when the 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 management of persistence, the specification of security, the definition of GUI (*Graphical User Interface*), the handling of software components, etc.
- if the model (dedicated to one of the concerns) is independent from the software platform on which the application will run, then it is possible to capitalize the know-how of the application independently from the context of use. Then the representation of the domain-specific knowledge is much more reusable.
- Generative Programming paradigm is a powerful approach which allows to compose all the different parts handled by those specific models in order to build the application according to the context of use (e.g. technologies related to software platforms).

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

Our approach relies on the concept of software factory [9]. First we give an overview of our approach, then we provide more details about some of the models that may be used during the design and development process of an application. In particular we provide details about the implementation of these models in the framework of the SMART-TOOLS software [31]. Finally we compare our approach with the state of the art.

2. Overview of the approach

We propose an approach that we think more adapted to the design and implementation of applications which rely on one data-model. It conforms to ideas developed in research works dealing with Domain Driven Development. It is mainly based on the ability to define business models and then to perform transformations on those models in order to generate either refinements of those models or platform-specific models.

We promote the idea that the use of this approach allows to get open, adaptable, and evolutive software. More precisely, thanks to generative programming, new programming paradigms and technologies can be easily integrated from the models into the target implementation programming language, at any time.

In figure 1 we aim to show the impact of our approach on the development of an application. Let us assume that to define an application we need to:

- define the data model associated with the application. It could be an information system, a programming language, an Object-Oriented method but also any other business model.
- specify the semantics of the model, that is to say the set of rules that manage all instances of the model and the properties that may be implicitly suggested by some of the model entities.
- design the architecture of an application as a set of software components that interact one with the others. Then it is necessary to be able to describe those components with their interactions.
- implement some orthogonal services such as persistence management, data integrity, distribution of objects over the network, GUI, etc.

According to those needs and to the key-ideas mentioned in section 1), we propose in figure 1 to address four models. Each of them is dedicated to the specifications related to one concern: the data model, the semantics, the set of components and an additional (orthogonal) service. Of course an application may rely on other(s) service(s). Depending on

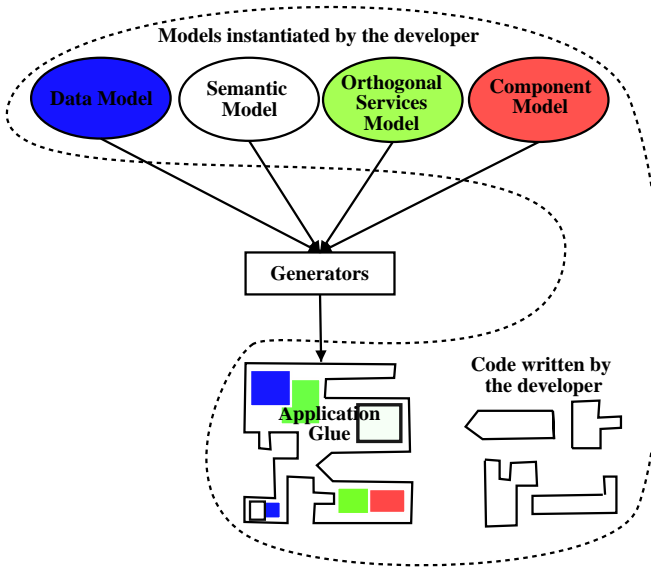


Figure 1. Our approach applied to the development of an application

the application needs it will be a replacement or an addition of service(s).

Thanks to the use of domain-specific and platform independent models (PIM), the business model is specified by parts which are independent from the platform on which it will be applied: persistence and security could be handled by any database management systems (DBMS), the model of components used may be EJB, CCM from OMG or any other (like the Web services from W3C), the language used to implement the model or its semantics may not be known at design time.

The generators associated to those models will then handle the generation of the application, providing the glue which is necessary to make it working on a specific platform, according to the context of use. If the platform changes or if the technology of the platform evolves, it is not necessary to update the models which represent the domain-specific know-how, but the generators only. In the next sections we intend to give more details on the four models that we consider as mandatory in the development process of an application:

- 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 *semantic model*. It integrates some facilities in order to structure and to modularize the code. This should help to maintain it and to facilitate its reuse;
- the *component model*. It is as close as possible from

the application needs. In particular it allows to specify the provided and required services.

- the *GUI model*. It define several views of the data model such as a structured editor in order to create and update more easily instances of this model; they must be device-independent.

In order to convince the reader with the interest of separating the concerns through different models, we propose a more detailed description of those models and additional explanations about the handling of those models in SMARTTOOLS². The experience of SMARTTOOLS aims to *i)* give a more precise description of the approach and *ii)* show how the approach favours the possible adaptations of an application according to the future evolutions of the software platform. In order to summarize, let us come back on the key-aspects of our approach.

First, it follows the MDA approach (see figure 2). In particular it uses source code generation as one transformation mechanism in order to produce one or many Platform Specific Models (PSM), from the Platform Independent Model (PIM) such as those mentioned above. We claim also that the development process of open and adaptable applications should rely on several PIMs.

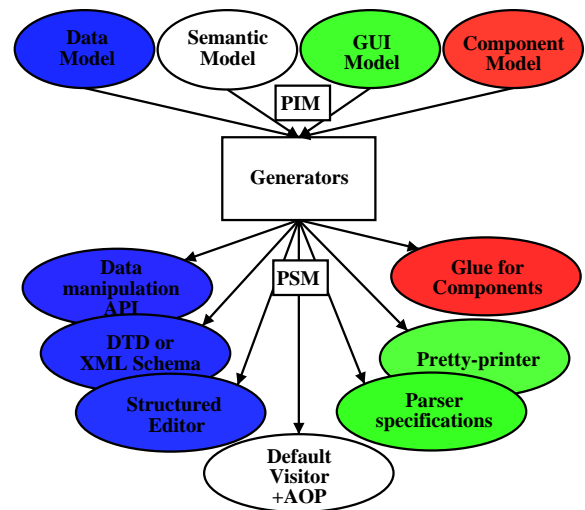


Figure 2. Generative Programming and MDA approach

Secondly our approach deals with the specification of a generative programming system which had been compared by Krzysztof Czarnecki and Ulrich W. Eisenecker to a factory of a particular domain [9]. It could be applied to any application which has a underlying data model.

²SMARTTOOLS is a research prototype whose aim is to validate our approach for different domains in an homogeneous way.

3. Data Models

In research works on programming languages, there are many formalisms to declare data structures (types). These formalisms are bound to either an underlying theory, associated systems (e.g. algebraic types for rewrite systems), or programming languages (e.g. ML language [22]). The most famous theory is certainly the notion of abstract syntax (very often mapped to a tree structure description) that contains the common concepts of those theories. Among them there is the concept of meta-language that may be represented by a BNF - *Backus Naur Form*.

For some years, the standardization efforts of both OMG and W3C play major roles in the data and model integration problematic. These standard formalisms continuously evolve in order to better address the new needs of applications. For instance, to improve document data validation, the document meta-language (DTD - *Document Type Definition*) has been replaced by more complex and rich data type formalisms such as XML Schema or RDFS (*Resource Description Framework Schema*). Another example deals with object-oriented modeling : the UML approach has evolved toward a domain-specific model definition based on the MOF (*Meta-Object Facility*) meta-formalism [12]. The databases have also evolved from relational databases toward object databases [5] and then toward XML databases [3].

Implementation: the experience of SMARTTOOLS.

Instead of using the formalisms mentioned above, we have preferred to define our own abstract data meta-model which is close to our needs and independent from any formalism. This meta-model aims to define the business models associated with the application. It has a leading role according to those various formalisms (DTD, XML Schema or MOF). For instance, we have defined translators (in both ways) between our meta-model and the DTD or the XML Schema meta-models. They make possible to import models in SMARTTOOLS as either a DTD, an XML Schema, or a document compliant to our meta-model. For example, this meta-model may be used to define the abstract syntax of existing programming languages as well as domain-specific languages; it is the cornerstone for all the generated tools and components. The goals of a such meta-model are the followings:

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

From this data model, SMARTTOOLS can generate, as shown by figure 2, following capabilities:

- *an API*. It helps to the manipulation of abstract syntax trees (for instance, in order to write semantic analyses);
- *an equivalent DTD or XML Schema*. It provides some facilities to the designers for exporting models;
- *an editor guided by the syntax*. It is a basic GUI that may be generated automatically in order to facilitate the handling of instances of the model.

In order to avoid the design and the implementation of another proprietary tree manipulation API, we have chosen the DOM (*Document Object Model*) API standard as tree kernel. In this way, the code which is specific to SMARTTOOLS for manipulating trees is minimal thus easy to maintain. Moreover our tree implementation benefits from any new service and bug fixing when this standard and its different implementations evolve. Thus 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 3), transformed with XSLT (*Extensible Stylesheet Language Transformation*), or addressed with XPath for free as these services are provided by the DOM API.

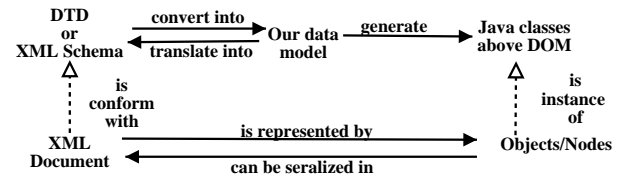


Figure 3. Bridge between languages and documents

One of the drawbacks about using the DOM standard is that it does not fulfill all our needs : the manipulated trees are not strictly-typed (a DOM tree has only homogeneous nodes). The main consequence is that it is difficult to semantically analyze them with a visitor-pattern approach (see section 4). The handling of strictly-typed trees may be achieved thanks to the generation of a language-specific API (Java classes) above the DOM API (type names of the nodes and the name of accessors are provided by the model).

4. Semantic Models

Let us coming back on the fact that one of our main goal is that the semantics which is associated to the model should

allow the model to be executable. Starting from this requirement we propose a pragmatic approach which rely on approaches applied to the specification of semantics within programming language and design methods. An interesting side effect of this choice is to reduce the gap between the design of an application and its implementation.

On the first hand, existing models like UML [28] or MOF [12] provide features for specifying the entities of a model as well as constraints on those entities; moreover works on action semantics [34] propose a graphical language for defining the body of methods but it seems that much remain to be done to make the model executable.

On the second hand, one of the approach which is applied to the implementation of the semantics of programming language within compilers is based on the description of pieces of code. They are written in the language which is used for the implementation of the compiler.

According to this background our approach wants to take the benefits from programming languages and OO design methods. First it includes a way to define the semantics of entities of the data model through a set of routines specified with one programming language which acts on the model entities (they are named: *actions*). The expressiveness of such approach is straightforward but it is quite crude. A more declarative and structured approach would help the designer of the model to describe the semantics.

Later in this section we propose an overview of our proposal for the definition of the semantics related to the data model. It takes into account some capabilities allowing to:

- split and compose the description of the semantics according to the different concerns related to the model and its instances,
- factorize the common semantics of the model entities by the use of meta-information and meta-assertion [8],
- specify better the different actions associated to model entities

A clear distinction should be made between people who describes the model³, and people who uses it and create instances of the model.

Which ASoC approach? According to what is mentioned above, the behavior of the model entities may deals with several distinct concerns. It may be interesting to rely on paradigms which handle the separation of concerns (ASoC). In the following, we propose a quick overview on the possible ASoC approaches that may be use to specify the semantics of a model.

New programming paradigms appear such as AOP [19], SOP [14] or Generative Programming [9]. In a certain way,

³They are experts of the domain and have to specify the know-how which is associated to it.

the GOF book on design patterns [11] is dealing with this problematic and proposes patterns to design applications wmore generic and flexible. One of the most famous design patterns is the *visitor*; it separates the data structures (a hierarchy of classes) and the associated treatments. Thanks to visitors, treatments are written in a modular way (one class) so that modifications and extension are easier to do.

The adaptable programming [23, 30] has extended the design pattern *visitor*. It provides a better flexibility towards changes related to the data model. More precisely, a traversal description language (a kind of extended pattern matching model) makes possible to cut off from the underlying data structure of the application.

The design pattern *visitor* has been the basis of many other research works [29]. It is close from the concept of multi-methods [4, 25] or of generic functions for functional programming. There are also other approaches which introduce genericity within programming languages. We may mention C++ templates, generic libraries [26], generic Java types [36] and programming by mixins [10]. All of them [27] need either, source code transformations, reflexivity, meta-programming, or higher order techniques, to work out.

All the mechanisms mentioned above are not sufficient and AOP [19] represents one of the most interesting answer. The main objective of this new paradigm, like the visitor design pattern, is to split up the application code into entities (class, module) associated with a concern in order to ensure a better application modularity.

To factorize and better capture semantics. According to what has been suggested above our approach provides a way to define the semantics of each entity through the use of a set of visitors that encapsulate all the concerns related to it. In next paragraph we describe how this approach is integrated in SMARTTOOLS but let us provide some details about new improvements that are undergoing specification:

- the use of declarative languages should be used in order to provide a support for the generation of the body of actions. As a first approach this is OCL (*Object Constraint Language*) that will be used and a classification of assertions will be made in order to better organize the specification of the semantics and to encapsulate additional information.
- the specification of actions should be improved in order to provide to the generator more information about the different concerns and the role of the actions; this should have an impact on the part of code of an action that can be automatically generated. The underlying know-how that we plan to use comes from the attribute grammar approaches [7, 18].
- we propose to include several sets of meta-information which will address some key-points of the semantics of

the most interesting entities of the model. Let us take the example of one business model whose aim is to describe the operational semantics of object-oriented languages [8]. Instead of trying to define by hand the semantics of all kinds of *classifiers*⁴ of all languages, we prefer to extract a set of properties which allow to characterize each kind of classifiers and to write generic actions which depend mainly on the value of those properties.

Thanks to these new capabilities we intend to improve the expressiveness of the meta-model in order to reduce the amount of code which has to be written when a model is described and to increase the impact of Generative Programming according to the specification of the semantics of a model.

Implementation: the experience of SMARTTOOLS.

In SMARTTOOLS, we have created a generator that instruments business models through the intensive use of the visitor design pattern. Thanks to the data model and pieces of meta-information about the semantic analysis (*visit* method signatures and traversal strategies), the generator can produce a visitor by default that visits only the nodes included into the traversal strategies. When the model designer wants to associate a semantic analysis to a model, he only has to extend by inheritance the visitor defined by default and override some of the *visit* methods in order to specify the accurate treatment.

In SMARTTOOLS, we have chosen to introduce AOP techniques which are designed especially for the meta-model which is dedicated to the description of semantic analyses (e.g. a type-checker or an evaluator). 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 analyses is performed through an extension of the generator which has to produce also the specific aspect-plugging code. It is embedded into the visitor defined by default. Thanks to this extension the semantics analysis attached to one data model can be extended, not only by inheritance, but also (dynamically) with aspects. 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 operational semantics is simple to understand, and *iii)* straightforward to implement so that potential evolutions of the needs of business models can be quickly integrated.

Let us now summarize and pay a particular attention on the implementation strong modularity. The handling of the semantics of business models is splitted into three distinct parts: the data model representation, the management of

its recursive traversal and the semantic actions (the treatment) to be attached. Our choice to introduce a separation of concerns even at the level of implementation favours easy model transformations and has an interesting impact on the ability of SMARTTOOLS to support applications of the MDA approach.

5. Component Models

The evolution of programming languages has also deeply changed the notion of modularity. Let us take some examples such as ADA or Java packages, generic libraries in imperative programming, multi-inheritance and contract approach in Eiffel [24] or the “module” in functional programming [22]. These different approaches provide interesting mechanisms in order to achieve more generic components but they are rather complex to use and they are not perfectly suitable for the needs of distributed applications. Many component technologies have been proposed in order to cope with these new needs: COM and DCOM for Microsoft, CCM for the OMG, and EJB for Sun. More recently, the Web-Services technology appeared with the possibility to list the component services in catalogs (UDDI - *Universal Description, Discover and Integration*).

According to the state of the art, three of the main challenges in component technologies are the followings:

- *to extend the classical method-call.* It allows to take into account the runtime environment (a three-tier architecture : the Internet, a message service, a database access) without any modification of the business code;
- *to extend the notion of interface.* It provides a way to describe/discover the provided and required services (e.g. the introspection in Java Beans) and to dynamically adapt the interface (e.g. 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, the security policy, etc.

These different mechanisms must be transparent towards the business code of the components. It corresponds to a kind of separation of concerns to avoid mixing the functional and the non-functional code. The OMG has proposed the MDA approach based on model transformations. It makes possible to get a better evolution of complex software applications towards component technologies [37]. This benefit explains the research efforts dealing with the definition of new generic component language and the links with both AOP and model transformations.

⁴In UML this is the name used to address the concept of data type

Implementation: the experience of SMARTTOOLS.

The aim of SMARTTOOLS is to define new data model and to import existing model representations. In order to achieve this objective it is mandatory to have a component architecture which guarantees an easy evolution of the set of models handled by SMARTTOOLS⁵. Moreover, having a component architecture (more precisely a meta-tool in the context of SMARTTOOLS) allows to build also an application with only the required components.

The first step were to define an abstract component model i.e. independent from any component technology. The issue is to be able to clearly identify the needs of SMARTTOOLS. Without this model, these needs would have been hidden by the use of a component format which is non model-specific (e.g. IDL - *Interface Definition Language*). Thanks to this component model, a generator can automatically produce the non-functional code, i.e. the container that hides all the communication and inter-connection mechanisms. For example let us take the broadcast mechanism which is used by a logical document and which aims to update the graphical views associated with it. This problem is totally transparent for the designer of application. Additionally, it is very easy to adapt the architecture in order to introduce a new communication mechanism.

The second step were to define a set of model transformations (projections) from our abstract component model towards well-known component technologies such as web-services, CCM, or EJB (see figure 4). It makes easier the exportation of the produced components.

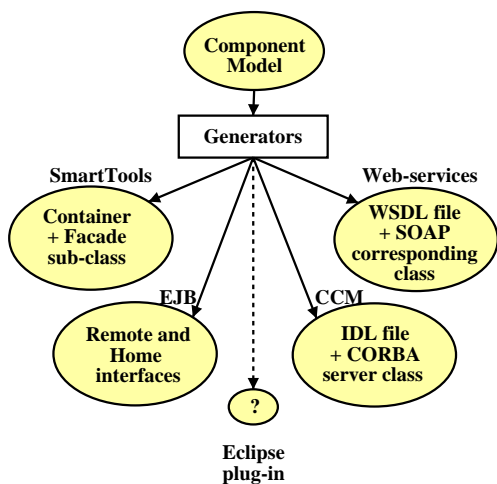


Figure 4. Component model transformations

Thanks to the experience gained by building projections, we do not believe that any of the three component technologies mentioned above (Web-Services, CCM, EJB) would fit

⁵For example it will be crucial to handle the possible inter-connections between models.

with our needs according to the component model itself and the specifications of connections between models. They are well appropriate for distributed applications but very poorly for applications with a generic (thus configurable) GUI. Moreover, the connection process proposed in SMARTTOOLS is much more flexible and dynamic than those offered by these technologies which are mainly dedicated to client/server architectures or Web servers. In SMARTTOOLS, component inter-connections are performed dynamically when it is required. It also applies a pattern-matching on techniques on the names of services which are provided or required by the components in order to bind the connectors.

Another important aspect of our component model is that it is strongly linked with the meta-model which describes data models. This means that the components may be built knowing the data-model representation. The main consequence is the ability to exchange complex information between two components such as sub-trees or path information (XPath) for the views and the associated logical document. The implementation of our model was rather easy.

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 implementations in different technologies (Platform Specific Models). In this way, our models and the generated ones can evolve and be much better adapted. The immediate result of this approach was to clearly identify the kernel⁶ of our tool.

Following our approach, the architecture of the produced applications are *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.

6. Graphical User Interfaces

The graphical interfaces that make applications interactive must also be adaptable to their evolutions. Two main challenges, when designing a graphical interface, should be kept in mind: the interface might be executed on different visualization devices and also be accessible 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.

⁶Only 0.5 Mbytes.

Implementation: the experience of SMARTTOOLS.

One of the goal 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. Thus it was mandatory to establish a bridge with the *Web semantics* research topics, and to provide tools which are easy to use and which are built on well-known techniques. According to this context, it is possible to quickly implement an environment dedicated to a business model which may have one or more specific views of the documents. These different displays, more user-friendly and more readable than the XML format are obtained through a sequence of model transformations or refinements (see figure 5).

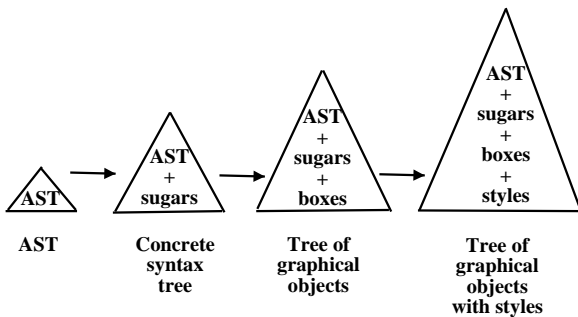


Figure 5. Graphical view obtained by successive model transformations

A GUI can be considered as a tree of graphical objects (windows, tabs, panels, views, menu, etc.). According to this remark, the approach based on visitors mentioned in section 4 can be reused in order to generate graphical views. In the same way, all the abstract syntax tree manipulation methods (insert a node, remove a node, etc.) but also the implementation to obtain a view can be reused. For example figure 6 shows three different views of the same GUI description document: the generic XML display on the right, the generic hierarchical tree display on the left, and the main window of application. We have defined a simple GUI-specific model useful to configure the GUI according to the applications.

7. Related Works

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 draw-

ing 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 qualities 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. 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 [1, 32]. In the same way, it exists other research works on the design pattern *visitor* which presents different variants of this design pattern [21, 15, 30].

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 usual component technologies mentioned earlier are mainly interested in the distribution concern.
- to remain on the implementation level. The UML modeling approaches [17, 12, 16] 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 [6, 20, 18], introduce a strong dependence between the generator and the produced code.
- to be evulative and open thanks to the use of standard mechanisms (e.g. XSLT for program transformation). For example according to program transformation, there are many other tools available [20] but with proprietary input formats and interpreter engines that require additional work to plug 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 the IDEs [2] that force the produced applications to be integrated into the IDE frameworks.

Finally, to make an exhaustive comparison with related works is almost impossible and is quite in opposition with the main objective of this paper. Indeed, as our approach gathers different domains, it exists, for each domain, many tools and research works related to our work that we should refer. We do not look for the best techniques for each concern but instead we want to show how different techniques can be integrated in a factory in order to obtain open and evulative application.

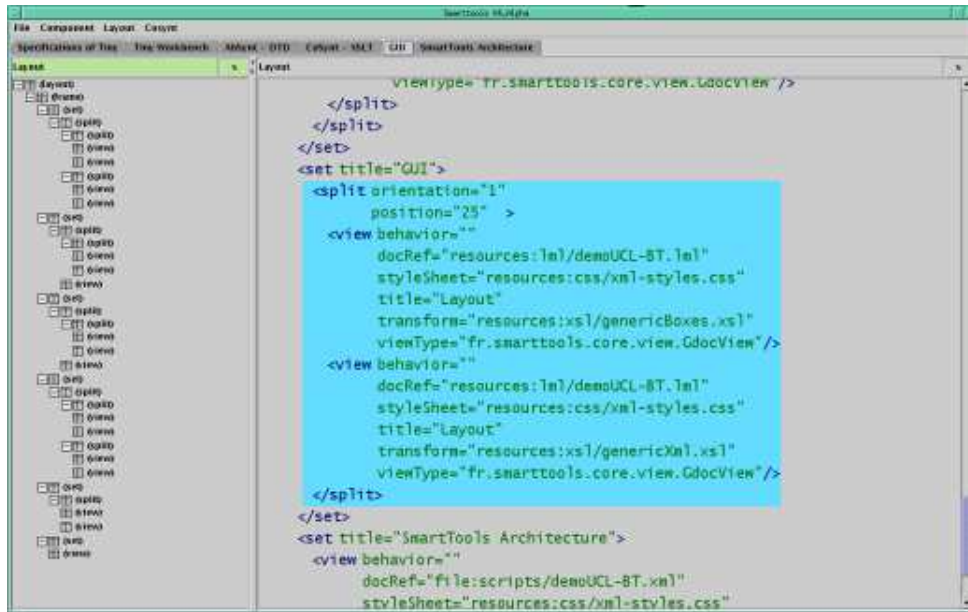


Figure 6. An example of SMARTTOOLS GUI.

8. Conclusion

Trough the continuous development of SMARTTOOLS, we are validating a new approach of 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. 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 thanks 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 generator associated with each models (data model, model of one specific service, component model, GUI model, etc.). These generators contain the design methodologies (they represent the software tool factory), they are customized thanks to input models, and they produce new intermediate models (they represent refinements) or the final models adapted to the software platform.

References

- [1] AspectJ - Aspect-Oriented Programming (AOP) for Java. <http://www.eclipse.org/aspectj/>.
- [2] Eclipse. <http://www.eclipse.org/>.
- [3] S. S. Abiteboul, P. Buneman, and D. Suciu. *Data on the web: from relations to semistructured data and XML*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 1999.
- [4] R. Agrawal, L. G. DeMichiel, and B. G. Lindsay. Static Type Checking of Multi-Methods. In *Proceedings of the OOPSLA '91 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 113–128, Nov. 1991. Published as ACM SIGPLAN Notices, volume 26, number 11.
- [5] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. In *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, pages 223–240, Kyoto, Japan, Dec. 1989.
- [6] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: the System. *SIGSOFT Software Eng. Notes*, 13(5):14–24, November 1988.
- [7] L. Correnson, E. Duris, D. Parigot, and G. Roussel. Declarative program transformation: a deforestation case-study. In G. Nadathur, editor, *Principles and Practice of Declarative Programming PDP'99*, volume 1702 of *Lect. Notes in Comp. Sci.*, pages 353–369, Paris, France, Oct. 1999.
- [8] P. Crescenzo and P. Lahire. Customisation of inheritance. In *ECOOP'2002 (The Inheritance Workshop)*, pages 23–29. University of Jyva"skyla", Finland and Workshop abstract to appear in LNCS, June 2002.
- [9] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Techniques, and Applications*. Addison-Wesley, June 2000.

- [10] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In ACM, editor, *Conference record of POPL '98: the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the Symposium, San Diego, California, 19–21 January 1998*, pages 171–183, New York, NY, USA, 1998. ACM Press.
- [11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, 1995. ISBN 0-201-63361-2-(3).
- [12] O. M. Group. Meta Object Facility (MOF) specification (version 1.3). Technical report, Object Management Group, Mar. 2000.
- [13] O. S. S. Group and R. Soley. Model-Driven Architecture. Technical report, OMG, November 2000.
- [14] 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.
- [15] G. Hedin and E. Magnusson. JastAdd—a Java-based system for implementing front ends. In M. van den Brand and D. Parigot, editors, *Electronic Notes in Theoretical Computer Science, LDTA'01 First Workshop on Language Descriptions, Tools and Application, ETAPS'2001*, volume 44, Genova, Italy, April 2001. Elsevier Science Publishers.
- [16] J.-M. Jézéquel, W.-M. Ho, A. L. Guennec, and F. Pennaneac'h. UMLAUT: an extendible UML transformation framework. In R. J. Hall and E. Tyugu, editors, *Proc. of the 14th IEEE International Conference on Automated Software Engineering, ASE'99*. IEEE, 1999.
- [17] 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.
- [18] M. Jourdan, D. Parigot, C. Julié, O. Durin, and C. Le Bellec. Design, Implementation and Evaluation of the FNC-2 Attribute Grammar System. In *Conf. on Programming Languages Design and Implementation*, pages 209–222, White Plains, NY, June 1990. Published as *ACM SIGPLAN Notices*, 25(6).
- [19] 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.
- [20] P. Klint. A Meta-Environment for Generating Programming Environments. *ACM Transactions on Software Engineering Methodology*, 2(2):176–201, 1993.
- [21] T. Kuipers and J. Visser. Object-Oriented Tree Traversal with JJForester. In M. van den Brand and D. Parigot, editors, *Electronic Notes in Theoretical Computer Science*, volume 44. Elsevier Science Publishers, 2001.
- [22] X. Leroy. Manifest types, modules, and separate compilation. In *POPL'94*, 1994.
- [23] K. J. Lieberherr and D. Orleans. Preventive Program Maintenance in Demeter/Java. In *Proceedings of the 19th International Conference on Software Engineering*, pages 604–605. ACM Press, May 1997.
- [24] D. Mandrioli and B. Meyer, editors. *Advances in Object-Oriented Software Engineering*. Prentice Hall, New York, 1992.
- [25] T. Millstein and C. Chambers. Modular statically typed multimethods. In R. Guerraoui, editor, *Proceedings ECOOP'99*, LCNS 1628, pages 279–303, Lisbon, Portugal, June 1999. Springer-Verlag.
- [26] D. R. Musser and A. A. Stepanov. Algorithm-oriented generic libraries. *Software Practice and Experience*, 24(7):623–642, July 1994.
- [27] M. Odersky. Objects + views = components? *Lecture Notes in Computer Science*, 1912:50–68, 2000.
- [28] OMG. UML - Unified Modeling Language. <http://www.uml.org>.
- [29] 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.
- [30] J. Palsberg, B. Patt-Shamir, and K. Lieberherr. A New Approach to Compiling Adaptive Programs. In H. R. Nielson, editor, *European Symposium on Programming*, pages 280–295, Linköping, Sweden, 1996. Springer Verlag.
- [31] 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).
- [32] 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.
- [33] C. Simonyi. The death of programming languages, the birth of intentional programming. Technical report, Microsoft, Inc., Sept. 1995.
- [34] G. Sunyé, F. Pennaneac'h, W.-M. Ho, A. L. Guennec, and J.-M. Jézéquel. Using UML action semantics for executable modeling and beyond. In *Advanced Information Systems Engineering. 13th International Conference, CAiSE 2001, Interlaken, Switzerland, June 4-8, 2001, Proceedings*, volume 2068 of *LNCIS*, pages 433–447. Springer, 2001.
- [35] C. Szyperski, J. Bosch, and W. Weck. Component-oriented programming. *Lecture Notes in Computer Science*, 1743:184–192, 1999.
- [36] P. Wadler. GJ: A Generic Java. *Dr. Dobb's Journal of Software Tools*, 25(2):23–26, 28, Feb. 2000.
- [37] 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.