Toward a New Approach for the Development of Software: the Model-Oriented Programming

Philippe Lahire[1], Didier Parigot[2], Carine Courbis[3], Pierre Crescenzo[1], and Emanuel Tundrea[4]

[1] Laboratoire I3S (UNSA/CNRS), 2000 route des lucioles BP 121,
F-06903 Sophia-Antipolis CEDEX, France
{Philippe.Lahire, Pierre.Crescenzo}@unice.fr
http://www.i3s.unice.fr
[2] INRIA Sophia-Antipolis 2004, route des Lucioles - BP 93
F-06902 Sophia-Antipolis CEDEX, France
Didier.Parigot@inria.fr
http://www.inria.fr
[3] University College London, Computer science department, Adastral Park,
Martlesham IP5 3RE - United Kingdom
carine.courbis@bt.com
[4] Politehnica University of Timisoara,
Faculty of Automatics and Computer Science, Bd. V. Parvan no 2,
1900 Timisoara, Romania
emanuel@emanuel.ro

Abstract. Nowadays, companies involved in the development of modern software face several difficulties. One of the most important ones is the continuous evolution of software platforms (C++, Java, DotNet, CORBA, EJB, Web services, XML, etc.). One interesting solution to this problem is the Model-Driven Architecture (MDA) approach from the OMG. It suggests that domain-specific knowledges should be encapsulated in platform-independent business models, apart from the applications. Beyond this answer is the failure of classical development techniques that rely on object-oriented design and programming. According to these remarks, we propose another way to develop software: *Model-Oriented Programming*. It is based on the Domain-Driven Development track and introduces a macro-level on top of the classical programming entities. It intends to be used for the handling, reuse and evolution of the business know-how and its associated applications. This paper *i)* attempts to define a set of golden rules for setting up the framework of model-oriented programming and ensuring the success of its use; *ii)* introduces a validation of those rules which relies on object-oriented and component-oriented technologies, MDA, aspect-oriented and generative programming; *iii)* addresses implementation issues of a prototype which benefits from earlier experiences.

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 Internet that implied applications no longer stand-alone 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 is difficult to choose the right and more capable of evolving one. For instance, to obtain a component-based application, a developer must choose between, at least, three component technologies: CCM (*CORBA Component Model*), EJB (*Enterprise Java Bean*), or Web Services.
- The third reason is the democratization (widespread) of computer science. 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-cuts 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 emergence of new programming paradigms such as AOP (*Aspect-Oriented Programming*) [17], SOP (*Subject-Oriented Programming*) [15], IP (*Intentional Programming*) [28], or component programming [32]. At the specification level, a strong and continuous evolution is undergoing toward 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 MDA (*Model-Driven Architecture*) approach [33,5,14].

According to these observations, we propose another way of developing software, named *Model-Oriented Programming*. It is based on the Domain-Driven Development track (DDD) [9], which relies on several paradigms such as object-oriented technology, languages for components, MDA, approaches for the separation of concerns, and generative programming [8].

This proposal relies on previous works which deal on the one hand with metamodeling [7], and on the other hand with the design of a software factory called SmartTools [2]. It intends to enrich both approaches in order to make easier the development of domain-specific applications.

The main objective of this approach (called SmartModels) is on the one hand, to clearly identify, thanks to a meta-level, the semantics of concepts used for the modeling of a given domain, and on the other hand, thanks to approaches by separation of concerns and generative programming, to equip, in a modular way, the applications related to this domain.

This approach is original and may be distinguished from other approaches by the following characteristics: *i)* it introduces on top of the entity which structures the model (reification level), a semantical layer which enables to define and factorize the basic functionalities related to the domain, *ii)* it provides a set of facilities (in order to quickly build applications related to the model), which strongly relies on the two levels of the model (data and semantic models), *iii)* it ensures a clear separation between the model and the technologies which makes the model executable by a software platform.

The main interest of such an approach is to provide the power to define the semantics of the entities which are addressed by a model, independently from any application. In general, the semantics is spread out in the applications which may directly handle the model.

Our approach does not make any difference between the modeling of the business model and the modeling of its applications. Thanks to the semantics which is encapsulated in the entities, related applications may handle directly this knowledge without going through some implementation phases (the generation process takes care of this).

Finally, contributions of both generative programming and separation of concerns are used in order to achieve a better flexibility and modularity of the applications related to the model.

Section 2 attempts to define a set of golden rules for setting up the key-aspects and more generally the framework of model-oriented programming. Sections 3 and 4 provide an overview of SmartModels, our approach, and explain how it fits with the rule requirements. Section 5 deals with the validation of the approach; it draws the outlines of the prototype (SmartFactory) which implements SmartModels. Finally, Section 6 presents some related works.

2 Proposition of Model-Oriented Programming Rules

Model-oriented programming is a new approach for the development of software which takes advantage not only from object-oriented and aspect-oriented paradigms but also from information systems. As it has been mentioned above, model-oriented programming introduces a new level of abstraction (the model) which acts as an autonomous entity that may receive queries from satellite applications. The specification of both the model and the applications may use for example object-oriented and/or aspect-oriented approaches.

Each application is built around at least one business model. We address the most frequent case where one business model is predominant. Having a predominant model on which are plugged in different concerns of an application

is very similar to approaches by separation of concerns (ASoC). However, in model-oriented programming, concerns are attached to a business model instead of being weaved into object-oriented applications which may be executed with or without these concerns. The model has its behaviour (its semantics), but it does not invoke itself any treatment. On the contrary, the semantics of the model is addressed only when applications query the model entities in order to match their requirements. In the context of DDD, a business model may support two main categories of applications: i) those dedicated to the computation and/or the update of information recorded by the instances of models; their methodology is close from information systems, and ii) those which deals with the transformation of the model and which are particularly relevant in the context of MDA.

Model-oriented programming is definitively very different from other paradigms such as object-oriented programming (OOP). It breaks the supremacy of programming languages: the model is now the key-point whereas the formalism used to describe its instances play minor roles. This is the consequence of the collaboration between MDA and generative programming. Altogether these two paradigms contribute to link the model and its formalism(s), and this favours the coming out of Domain-Specific Languages (DSL) as business-models.

A new approach for the development of software must ensure that software engineering skills are covered and improved in comparison with object-oriented and aspect-oriented approaches. Reusability, evolution capabilities and robustness of both business models and applications must be addressed very carefully by model-oriented programming. We propose nine rules which characterize, from our point of view, model-oriented programming; they are classified in two categories: conceptual and implementation purposes. We take this opportunity to discuss how they address those issues.

2.1 Rules for the Approach Design

Rule N° 1: Business Model as a first-class entity of the development process. A business model relies on a data model and on a semantic model. The data model contains the description of the entities involved in the business model whereas the semantic model describes the interactions and the constraints between those entities, but also their behaviour with respect to the business-model know-how. A business model is considered by applications as a whole or for its contents; it constitutes a new level of abstraction which favours global operations such as transformation or introspection. Both of them query the model entities in order to reuse its business know-how or to involve both model and programs.

Rule N° 2: A triple independence between the model, the application and the technology. A business model is not an application. It encapsulates the description of its behaviour (its semantics), which must be independent from any further use. This property will ensure that a business model is reusable independently from

the applications that may address it[5]. Moreover, an application or a business model must be designed independently from the software platform on which the application will be executed. This is only at the very last moment that the binding with the platform technology[6] must be made. This second property allows the business logic to be used whatever technology will appear in the future.

Rule N°3: Support of generic entities. Typically business models may address lines of products and more generally a set of entities that may have commonalities and differences but which have a close semantics[7]; they must be designed as generic entities which may be easily derived. A quite common situation is that business models address a few key-entities which are defined according to a large number of basic entities; very often, the key-entities correspond to generic entities. Then it is particularly important that generic entities provide a clear vision of their semantics because they deal with a significant part of the model semantics. Object-Oriented languages like Eiffel proved that the support of generic entities (we should say generic business model) is an interesting approach to ensure reusability and evolution.

Rule N°4: Clear separation between semantic and data models. The domain-specific know-how is encapsulated in business models through the data model (reification and structuring by the entities) and the semantic model (behaviour of those entities). To be able to reuse the semantics when the data model evolves is an important issue. This is particularly important in the context of model transformations where semantics must evolve accordingly to the data-model (in the most automatic way). Model-oriented programming must provide a clear separation between the description of the data-model and the description of its semantics.

Rule N°5: Orthogonal handling of concerns. Rule N°2 infers a separation of concerns between the business model and the applications. The first one is under the responsibility of an expert which captures the domain-specific know-how, whereas the second one is handled by programmers. But separation of concerns must exist also within the business model and within the applications themselves.

According to the business model, the needs are twofolds: *i)* the semantics may be complex enough and require some modularization, and *ii)* pieces of semantics which are orthogonal to the original semantics must be straightforwardly carried out.

According to applications, the requirements are even more important. An application may contain different subjects which have to be smoothly composed for building it up. Moreover, an application should be able to take care about the

[5] Of course it is still an important issue to ensure also the reusability of the application behaviour.

[6] Some people call this the Platform Dependent Model (PDM).

[7] It is important to note that commonalities and differences may represent a major part of the semantics of these entities.

evolution of the environment (which can not be foreseen in advance), without changing the application core.

2.2 Rules for the Approach Implementation

Rule N°6: An adequate balance between declarative and imperative programming. Semantics of business models should be described as much as possible in a declarative way in order to specify what is expected (the "*what*") but not how it is made (the "*how*"). This is one of the most important issues addressed by the MDA approach. But, it is not acceptable to carry this approach to the breaking point where the description relies on very complex formalisms, difficult to read and to understand. A compromise is necessary between the "all declarative" and the "all programming".

Rule N°7: Support of domain-specific languages. A clear distinction has to be made between the expressiveness of a business model and the language (textual, graphical, etc.) used by the designer for the specification of the different pieces of this business model. Moreover, model-oriented programming tends to come closer and closer to the general public (ubiquitous programming), so that the need to provide "languages" dedicated to one business model and even to one application becomes more and more important. Generative programming and MDA provide a good support to achieve this issue.

Rule N°8: Openness of the development process. To provide a meta-model and a set of related mechanisms that answer to any need of any kind of business model is Utopian from our point of view. We promote the idea of an unified approach with very few built-in mechanisms, but that can be easily adapted to further needs of modern applications. In particular, it is important to be able to customize the way to query information according to the context of use. In other words, the generation and handling of an executable business model must be customizable. In our approach, all the key-concepts which participate to the description of both the application and the business model in order to make it executable are first-class entities.

Rule N°9: Self-extensible capability of the approach. Model-oriented programming requires a meta-model which captures the description of both business models and applications, as it is mentioned in previous rules. This meta-model may be considered as a particular business model. As it is explained in Rule N°7, the specification of the different parts of this meta-model may rely, for example on a dedicated language[8]. But many other needs required for the development of applications may appear. In particular, modern applications should be available as components that may interact one with the others. It is important to make the approach self-extensible, that is to say able to include other applications

[8] It can be built as a pseudo-language or it can use the UML graphical approach with activity or class diagrams, Action Semantics, etc.

and business models built thanks to model-oriented programming (that means built with the approach itself). For example, to handle components, a correct approach would be to design a business model.

With those nine rules, we attempted to set a framework for model-oriented programming. We promote the idea that an approach which intends to implement model-oriented programming should try as much as possible to match the requirements proposed by those rules. In the following sections we propose our approach and we demonstrate how it addresses the rules.

3 Key-aspects of business models with SmartModels

This section deals with the expressiveness of the meta-model which is dedicated to the description of business models. It addresses both data and semantic models with respect to the proposed rules.

As a preamble, we can say that the meta-model which allows the description of business models addressed *i)* the reification of basic entities (Section 3.2), *ii)* the reification of generic entities and their generic parameters (Section 3.3), *iii)* the semantics of the business model which corresponds mainly to the possible values that may be assigned to the generic parameters (Section 3.3) and to the actions (Section 3.4). One of the key-aspects of SmartModels is that the semantic model is encapsulated in a meta-level, so that it may be distinguished from the data-model. Section 3.1 explains the main benefits that are expected from this. Figure 1 illustrates these previous lines.

3.1 A meta-level to separate semantic and data-models

This section addresses principally the fourth rule. In Figure 1 we propose an overview of the architecture of the meta-model. The semantics of the business model is addressed through the specification of *hypergeneric parameters*[10], *characteristics* and *actions*. All of them participate to the definition of the semantics of business-model entities[9] (whether they are generic or not); but they do not address the description of their instances. Because applications are outside the business model, the methods that handle instances of atoms are accessors only[10]; they are automatically generated taking into account the type (for example if it is a collection or not). *Actions* are methods with special properties; for example, they can handle assertions (see section 3.4), aspects (see section 4.2). Moreover, actions are first-class entities (see section 4.3).

We propose to create a meta-level in order to encapsulate the semantics of an entity into a meta-level which is named the *concept*. A concept is associated

[9] We call them *atom* - see section 3.2.

[10] This is the main difference with actions which address entities but not instance of entities.

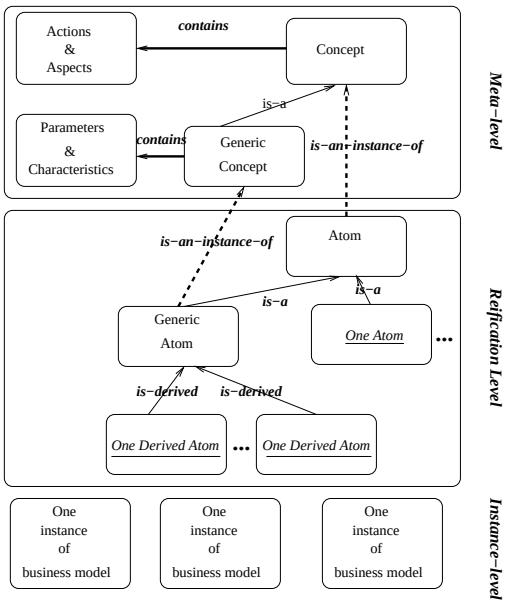

Fig. 1. Key-aspects of a business model

with one or several atoms[11]. This clear separation between the semantics of the business model and the reification of its entities is very important because it favours *i)* the maintenance of the semantics (redefining the semantics should only deal with concepts), *ii)* the reuse of the semantics in other (closely-related) business models, and *iii)* the transformation of model which is one of the key-points of model-oriented programming. MOF does not integrate any meta-level. The main consequence is that it is not possible[12] to indicate that a MOF class is an instance of another one. The main facilities provided by MOF to describe meta-information are class variables and class methods; from our point of view it is not sufficient.

3.2 Expressiveness of the data-model

This section addresses mainly the first and the second rules; it participates to the description of the data-model which is part of a business model.

The description of the business-model entities relies on well-known concepts that may be found in most programming languages or meta-models. We present them briefly in the context of SmartModels and with regard to MOF [13]. At

[11] It is an approach which is quite similar to the classes and meta-classes of the Smalltalk language.

[12] It is of course possible to write two models with MOF, one being the meta-model of the other. But according to our knowledge is not possible to express it with MOF.

a first glance, we could define business models directly with MOF, but Sections 3.1 and 3.3 demonstrate that additional information must be inserted.

In our meta-model, an *atom* is the structure which supports the description of an entity; it is very close to the MOF "*class*" notion[13]. Then the features provided by MOF to describe the contents of a class (such as attributes, operations, generalisation relationships) are sufficient to define most of the reification of an entity. MOF provides also the possibility to describe associations. To describe associations, we have introduced a generic type which implements different kinds of collection such as *bag*, *set*, or *list* (with or without bounds)[14].

The designer of a business model may create atoms either for improving the structuring and factorization of information within the model hierarchy, or for describing atoms which have instances within applications. Our meta-model provides a way to address those two issues; MOF does it through the notion of *abstract class*. If it means that the class must have at least an abstract method or that all the methods must be abstract, then we believe that this mechanism is not sufficient. In particular, some applications may be interested by some atoms whereas others are not; it is not the same thing to say that whatever is the context of use, one atom may not have instances because it is only partially defined. We believe that a more accurate information according to the atom status will improve the readability of the code produced by generators, and the facilities that may be provided or not to the programmer of application according to it. The interest to be able to associate different status with an atom is even greater if the business model may import atoms from another business model.

3.3 Support of generic entities

This section addresses mainly the first, third and fourth rules. It participates to the description of the data and semantics models; it addresses especially the handling of the key-entities of a model.

The support of generic entities (*generic atoms*) is an important issue for business models. Let us take an example of one business model which is dedicated to record both the structures and semantics of Java programs. Possible applications with respect to this model may implement functionalities of programming environments (metrics, various wizards or editors, etc.). Possible atoms of this model represent, for example, *attribute*, *method*, *method parameters*, *modifiers*, etc. But the most interesting ones deals with the different kinds of *classifiers* and *relationships* (aggregation-like or inheritance-like). Most semantics may be encapsulated within classifiers and relationships and other atoms mentioned above may have a very minimal semantics mostly represented by their reification. This is possible because they are driven by the semantics associated with classifiers

[13] The concept of *class* is, from our point of view, too much related to programming languages whereas business models require a more abstract concept.

[14] The MOF associations provide more capabilities but we are not sure at the moment that business model description requires it.

and relationships. In fact, there are several kinds of classifiers (e.g. *class*, *inner class*, *interface*, etc.) and relationships (e.g. *extends* between interface, *extends* between classes, *implements* between one interface and one class) in this business model [7]. Then it is meaningful to be able to record their definitions as generic atoms[15].
Let us define the term "*generic atoms*". The genericity comes from a set of *hypergeneric parameters* and a set of *characteristics* which records the differences and the commonalities between all the foreseen derived entities[16] (*e.g.* all the Java classifiers). The definition of an hypergeneric parameter is mainly based on a basic type (it may be an integer, a boolean, an enumeration, a tuple and a collection) and on some additional pieces of information. The definition of a characteristic relies on an atom or a collection of atoms (*e.g.* one kind of classifier records the possible kinds of inheritance-like relationships that it may declare). Intuitively, generic atoms are quite similar to the concept of generic class in the Eiffel language. But derived atoms are obtained through the relevant combination of values associated with the sets of characteristics and parameters which participate to the definition of the *generic atom*.

We choose to use generic atoms instead of inheritance relationships for modeling these atoms for several reasons: *i)* the definition of the data model is not mixed with the definition of the semantics; this increases the ability of the semantics of the business model to be transformed and reused in another model; *ii)* a significant part of the semantics of all the derived entities of one generic entity is recorded in one location, by the definition of its parameters and characteristics; this favours both (re)use and maintenance of these entities; iii) the reuse of a business model is improved, especially when the new model is an extension of the first one; according to the business model related to the Java language, to extend it with a new relationship like for instance, the reverse-inheritance requires only that the new business model describes a new instance of the generic entity which deals with inheritance-like relationships.

3.4 Description of the semantic model

This section addresses the first rule and more especially the description of the semantics model.

We explained in Section 3.3 that a significant part of the semantics of a business model is encapsulated in a few generic atoms. A part of the semantics is captured by parameters, characteristics and invariants[17]. It is a first step but it is still not sufficient to handle the full semantics of atoms. For example, the value of parameters used for the instantiation of generic atoms will affect the behaviour of its derived atoms. It is necessary to be able to specify this behaviour.

[15] One generic entity for modifiers, one for inheritance-like relationships and one for aggregation-like relationships.
[16] This is the term which is quite often used in the state of the art, to refer instances of generic entities.
[17] Like in MOF or UML, it is possible also to define atom invariants. This contributes to the description of the semantics of entities.

Each atom, whatever it is generic or not, has a meta-level (its concept) where it is possible to define actions; when the atom is derived from a generic one its execution is driven by the value of the parameters. For other atoms, the ability is provided but we believe that it is not relevant in most cases.

An *action* has a signature, preconditions and postconditions defined with respect to the reification of entities and hypergeneric parameters when it is a generic atom[18]. It may also accept the execution of orthogonal concerns (see Section 4.2). An action must be completely independent from the application related to the business model. A typical scenario is that the behaviour of a given application relies on the semantic model, that is to say call those actions or query hypergeneric parameters.

It is straightforward that an action has a "body" that has to be specified; there are three main approaches to take its contents into account in the meta-model: *i)* to propose a full representation of the body which may correspond to the reification of some pseudo-languages, *ii)* to delegate to the description of the body to the underlying implementation language; the meta-model records only the fact that an action has a body, and *iii)* to propose a partial description of the body[19]. Typically for the first and third solution the body of the action will be partially generated, whereas in the second solution, the whole description of the body will be completely under the responsibility of the developer. At the moment our first prototype implements the third solution but the expressiveness of action bodies is going to be improved as far as interesting capabilities are found.

It is important to distinguish the capability of the model to record more or less the representation of the action bodies, from the description language which is provided to the user in order to describe it. This aspect will be discussed in section 4.4, nevertheless two solutions seem to be relevant: *i)* to take UML from OMG and to use diagram of activities and/or Action Semantics, *ii)* to design a domain-specific language for the semantics description. We have not evaluated seriously these two solutions yet. It is said that UML is the meta-model for the specification of business models (it is important to reuse existing standards), but we have also to remember that at the beginning UML was not designed for the definition of business models but for applications. An alternative to those approaches may be to increase the expressiveness of MOF with respect to the description of the semantics.

4 Support for the Design of Applications

Section 3 explained how our meta-model for the specification of business models matches the rules of section 2. This section addresses the description of applications which capitalize the atoms of the business model. As it has been mentioned earlier, we can distinguish two kinds of application: those which describe model transformations and those which query, compute and update the instances of

[18] In our meta-model, assertions are described with OCL from the OMG.

[19] For example, to record the list of hypergeneric parameters that are involved in the semantics of the action

the business model. At this point, it is straightforward that the specification of those applications will slightly differ from classical object-oriented applications, even if both rely on the object paradigm.

Intuitively, an application corresponds to the traversal of a graph of atoms represented by one business model. During this traversal, the behaviour contained in application facets are performed sequentially (see section 4.1). While these facets are processed, it is possible to trigger the execution of aspects which allows to integrate orthogonal services (see section 4.2). The reification of both the business model and the application is handled by a meta-object protocol (see section 4.3) which contains also additional functionalities.

4.1 Vertical Cross-Cutting

This section addresses the fifth, sixth, seventh and ninth rules. The vertical cross-cutting of applications is itself defined as an independent business model, so that it may also be associated with a DSL (see section 4.4). A *facet* regroups a set of entities called *visit entities*. Both of them work toward the description of program related to the business model. We define what the facets and the visit entities are.

What is a Facet? A facet represents one concern of the application with respect to the business model (see figure 2). This is a vertical cross-cutting of the application whereas inheritance relationship would provide an horizontal cross-cutting which introduces several levels of abstraction into the business model or the application[20]. The organization by facets of an application comes from subject-oriented programming (SOP). Each facet corresponds to a part of the treatment to be processed on one entity. Typically one facet of a given application would rather address the same set of atoms as the other facets (even if there is no constraint). The description of a facet relies on the design pattern *Visitor* [26,23]. Depending on the requirements of applications, the behaviour related to the visit of one atom may be spread out differently. In a first approximation, one atom of the business model is associated with only one visit-entity.

A facet specifies *i)* the business model that it addresses, *ii)* how the business model is visited, *iii)* the entities that are relevant according to the objectives, and possibly *iv)* some additional technologies. Technologies are defined independently from SmartModels (for example, by an API or a library of classes). They contain functionalities which allow to define more easily the application. For example, the DOM API is welcomed to manipulate XML representation of the business models. Other information may be added in order to generate visit-entities which fit exactly to the expectations of the programmer. Indeed the source code generation is essential to our approach because it allows him, to focus only on the visit-entities that are addressed by the facet but also to be assisted for the description of their behaviour.

[20] The model supports hierarchies of atoms, concepts, visit-entities, facets and more generally of any first-class entity.

According to the possible complexity of application requirements, we propose to encapsulate their handling into another dedicated business model. A first step should ensure that one application may be composed with one or several facets that will be executed sequentially, but this is not sufficient and a more sophisticated composition mechanism has to be specified. Moreover, facets could be also components and applications should be able to take them into account. The fact to rely on a domain-specific model favours the reuse and the evolution of application behaviours.

What is a Visit Entity ? We consider a visit-entity as being first-class (see Section 4.3). It implements an execution model and a mechanism for binding the business model, which are very flexible and evolutive. Each visit-entity is described by an *execute* method which contains not only the behaviour of the visit but also a mechanism for the description and the verification of preconditions and postconditions. Like in programming languages such as Eiffel [22], preconditions and postconditions are evaluated respectively at the beginning and at the end of the method; they determine whether the visit of one atom succeed or failed. A visit entity contains also a direct access[21] to the atoms or to their properties when they should be handled by *execute*. The advantage is twofold: to minimize the code to write and to hide the complexity of the business model and of its representation. For example, when a visit entity is generated the system knows which is the related atom. Then the visit entity may include a direct access to the current instance of the atom as well as the execution context. It provides also other services such as the set of instances of the atom or its meta-information.

Customization of Business-Model visits As it has been discussed in Section 4.1, the behaviour related to the visit of one atom may be spread out differently, according to the needs of applications. In particular, applications obtained by successive transformations of models (Platform Independent Model - PIM to Platform Specific Model - PSM), require a dedicated spreading of the visit entities.

We already mentioned that the business model is a passive structure on which facets and their possible aspects are processed. The structure traversal is automatically generated; it starts from a given atom that may be specified by the application or the facet; by default, it is the root atom of the business model. This traversal corresponds to a navigation within the graph represented by all the properties of the atoms[22]. The atoms to be considered depend on the business model specification (they must have instances or being "visible") and on facets (a facet selects only atoms which are affected by its behaviour).

[21] It is automatically generated according to the information mentioned in the description of both a facet and a business model.

[22] Depending on the research community which is addressed, this graph may be called: client-supplier graph, aggregation or composition graph, delegation graph, etc.

At the moment, we propose three categories of visit-entities and a deep-first traversal[23]. The order used to "visit" atom properties is by default not meaningful and depends on the implementation, but for each atom it is possible to set one specific order.

Because of application requirements, some facets should be able to adapt the traversal of an atom and even decide to not traverse some of its properties. When a programmer needs such facilities, it is mandatory to show him the traversal, so that he may control the navigation within the atom subgraph. In such a case, the visit-entity will contain the description of the traversal. It is costly for the programmer as the complexity of the visit-entity increases but he will be able to insert any additional piece of behaviour into the traversal. The use of this category of visit-entity is not recommended when the semantics of application must be reused on atoms which evolve (e.g. they gain or loose some properties). For other applications, the semantics is independent from the graph traversal handling, and it is useless to show it to the application designer. Such visit entity addresses only the behaviour of the visit. This category of visit-entity increases not only the readability but also the reusability of the application semantics. Nevertheless, it remains very dependent on the atom structure.
Finally, when the need of flexibility is higher (this is typically the case when the business model may be involved in the transformation process of model instances), it is interesting to be able to easily transform not only the structure of the business model (the data-model), but also (and most of all) the behaviour of its related applications. To achieve such an issue, it is necessary that applications use visit entities which make the visit behaviour as much independent as possible from the structure of the business-model atoms. At the moment, we propose a category of visit entity which matches quite well those requirements. The granularity of visit entities had been increased: one visit entity corresponds to one property of a given atom instead of one for the whole atom. In other words, there will be as many visit entities as properties in the whole business model. Intuitively, if, in a model obtained by transformation, one property is located in an atom not from the original model, it should be easy[24] to transform the visit behaviour in order to apply it to the new business model.

Next section demonstrates that thanks to AOP paradigm, it is possible to insert new concerns with respect to the semantics of the application. This is completely independent from the category of visit entity.

4.2 Horizontal Cross-Cutting

This section addresses the fifth, sixth, seventh and ninth rules. Programming by facet enables to insert additional behaviour *a posteriori* within a new facet which will be added to the list of facets[25]. But it is not possible to add easily some pieces

[23] Other kinds of visit and graph traversal should be proposed in the future, depending on the needs.

[24] In most cases this should be possible to be realized in standalone.

[25] A facet intends to be rather independent from the other facets but of course the execution order may influence the result.

of behaviour which are orthogonal to all facets. It will be performed at some points of the execution of one or several facets in order to, for example, check the validity of a constraint, load data, check access rights or trace a method-call. In order to satisfy those requirements, we introduce a new capability which is derived from AOP (see Figure 2).

Aspects and Visit Entities Each visit entity accepts a list of aspects; this allows in particular to attach a single aspect to the set of visit-entities of *i)* a given facet or, *ii)* all the facets of a given application. Of course, those are two particular cases; the set of visit-entities to which an aspect may be associated is completely free. The expressiveness related to the specification of this set depends only on the language dedicated to the definition of joint-points within the facets[26]. This language does not require the expressiveness proposed for example in AspectJ and does not address program structures but atoms. For example, in a business model, if only one visit-entity is associated with a given atom, then there will be only one joint-point per atom. But in the same way as it is proposed for AspectJ, it is possible to customize one visit-entity in order to integrate an aspect at different moments: before the invocation of the visit or when its execution starts or ends. It is also possible to distinguish two kinds of execution ends: the execution fails (one assertion is not satisfied) or the execution succeed (all assertions are satisfied). The expressiveness of the handling of aspects and in particular the description of joint-points should be handle by an independent business model which, like facets could be associated with a DSL; it is another example of self-extensible capability of SmartModels.

Aspects for Business Models In the previous section, we explained how aspects interact with the application semantics. Now, we address briefly their integration within the business-model semantics (which is independent from potential applications).

The Section 3 showed that in SmartModels the description of the model semantics relies on *i)* actions that are described in a *concept* (at the meta-level) and, *ii)* invariants for concepts or atoms and pre/post conditions for actions.

Aspects may be integrated within both assertions and actions; they are very similar to those which are dedicated to the visit-entities. Why may we need to equip the semantics of business models with aspects? A first answer is that the semantics of a business model may be complex enough to feel the need to separate its different concerns; this favours readability, maintenance and reuse. We do not propose a double cross-cutting of the semantics as for applications because it seems more useful to favour the use of assertions which is more relevant for the description of business-model entities and for their interactions. Moreover, the number of statements which are necessary to the description of the semantics is clearly much smaller than the size of statements dedicated to the description of facets.

[26] In AspectJ, an aspect-oriented language, a joint-point is a particular location in the program where an aspect may be integrated.

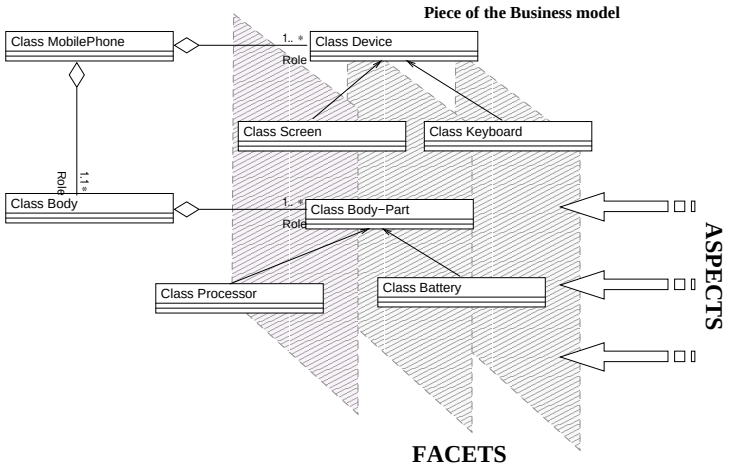

Fig. 2. Handling of application semantics by facets and aspects

At a first glance, it seems more interesting to add dynamically aspects to applications, whereas aspects associated with business models are more static. However when a model is obtained by transformation of another one it may be valuable to have the ability to adapt the semantics encapsulated within one action (proposed in the original model) in order to make it compatible with the new model. This is typically for this purpose that we provide aspects within assertions. For example, an aspect which implements an "around"[27] may decide that according to the context of execution, the assertion in the new model is still relevant or on the contrary, is not relevant anymore.

4.3 A Meta-Object Protocol

This section addresses in particular the seventh and eighth rules.

In previous section we addressed issues which are related to the expressiveness of the meta-model used for the description of business models. We underlined also the relevance of generative programming and model transformations. Our objective is to provide a way to program with a model-oriented approach to business model designers or application programmers. This means to be able to generate, reuse or transform any business model that can be specified by the meta-model and to write applications which address the capabilities of this model. An important challenge for us is to achieve these issues either automatically according to the declarative description of both business model and applications or to provide the most adequate help to the programmer to fulfill manually the remaining part of the description. We believe that the openness of

[27] An aspect "*around*" is executed before the invocation of *i)* an action, *ii)* a visit-entity or *iii)* an assertion. This suggests that we may not execute it.

the platform which supports the approach is a key-point for this purpose; we propose two techniques to increase openness: a meta-object protocol and first-class entities for the description of the semantics.

Actions, meta-information (parameters and characteristics), assertions, visit entities, facets and aspects are "first-class". That means, they are seen as any other entity of the business model. In particular, they may be extended with inheritance as any other entity[28]. Moreover the modification of the handling of those entities is straightforward.

4.4 Support of Domain-Specific Languages and Standards

This section addresses both seventh and ninth rules, and suggests that having user-friendly interfaces adapted to the business model is an important issue.

In SmartTools [2], we proposed a small language called *cosynt*[29] which associates syntactical or graphical items with one entity of a given data model, so that it generates a user interface dedicated to the model. Thanks to this approach, it is possible to address the input/display of the business model information with a more user-friendly interface. *Cosynt* relies on a business model dedicated to the specification of the GUI.

We intends to refine and improve both its syntax and business model in order to generate textual and graphical interfaces for SmartModels. In particular, the GUI business model must provide capabilities for the description of the semantics[30] which, at this moment, is not taken into account; only the data-model is.

The capture of external models is also an important issue in order to widen the set of user-interfaces proposed to both application programmers and business-model designers; the more efficient approach is to provide a path with standards of OMG and W3C. This is achieved thanks to the generation of a parser which handles business model described with XML. This opens SmartModels to all the models which propose an XML representation.
Thanks to both the GUI business model and the XML-dedicated parser, the description of the semantics may be handled by *i)* a pseudo programming language dedicated to the description of model-oriented applications and actions[31]; ii) description facilities coming from standards, like Action Semantics [31] or the activity diagrams proposed by UML [16], or iii) a language for the transformation of models.

[28] This suggests that the platform which describes the approach may be bootstrapped.
[29] Cosynt stands for "concrete syntax".
[30] For both the business model semantics and the application semantics.
[31] At this time, the language proposed by our prototype is the same as its implementation language: *Java*.

5 SmartFactory: An Implementation Centered on Models

In this section we provide some elements about how to develop with SmartModels (section 5.1). Then, we propose an overview of SmartFactory which is an implementation of SmartModels.

5.1 About the SmartModels methodology

The description of the business model is a five-step process: *i)* to identify and to specify the basic atoms of the model, *ii)* to identify the generic atoms, *iii)* to define the criteria of genericity (the hypergeneric parameters)[32], *iv)* to specify the actions attached to generic and none generic atoms, and *v)* to specify the instances of the generic atoms (derived atoms). The three last steps deals with the specification of the meta-level (the concepts).

The development of an application is also processed in several steps: i) to identify the facets (or concerns) of the application (they are all related to the same business model which is considered as the backbone of the application), ii) to detect the concerns that are enough general to be defined with another business model[33], iii) to find the possible technologies to use for the implementation and to specify their integration, iv) to define the contents of the visit-entities, v) to build the application with respect to facets and domain-specific components, vi) to think about possible aspects to weave with the application, vii) to describe the DSL attached to the business models which had been specified.
It has to be quoted that SmartModels applies to itself, so that we specified business models dedicated to the specification of facets, aspects, applications and components. The latter point is out of the scope of the paper but we consider it as an important issue: a set of treatments (a facet or an application) may be viewed as a component which provides some services and which may communicate with other components[34]. Each dedicated business model may be associated with a dedicated language (textual or graphical) in order to handle a user-friendly input/display of the information required by them.

5.2 Overview of SmartFactory

For a better understanding of the interest of SmartModels, it is important to give an overview of the implementation of its prototype called SmartFactory. We will address all the main aspects, from the description of the meta-model to the achievement of an operational system. Figure 3 describes the implementation

[32] Typically this is a step that must be perform by an expert of the domain. It represents a part of the knowledge of the business model.
[33] They will be integrated in the application as domain-specific components. This step favours the reuse of these concerns.
[34] This approach had been introduced in SmartTools.

of this prototype which is based on the SmartTools technology which represents the first step of the research conducted in the DDD framework[35]. Thanks to it, we can prove the feasibility and the interest of our approach, but also we can get some feedbacks in order to improve it.

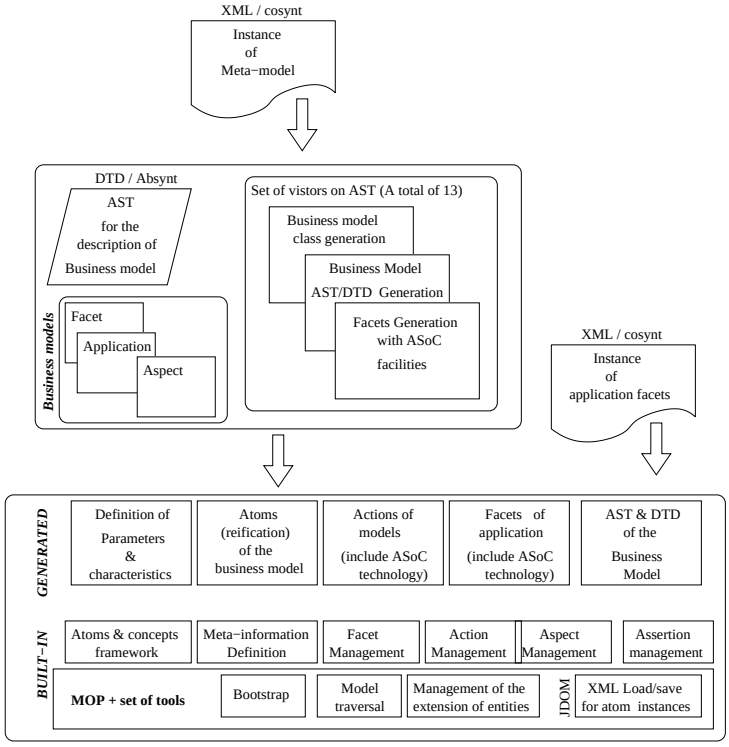

Fig. 3. Implementation of our approach with SmartTools

The meta-model whose key-aspects are presented in Section 3 is described with the *Absynt* language which is close from the BNF. From a *Absynt* model instance, SmartTools automatically generates *i)* a reification of the corresponding abstract syntax tree (AST) in Java *ii)* an equivalent DTD, of this AST, *iii)* a class which contains the set of visit methods associated with the entities of the data model (implementation of the design pattern *Visitor* [26]). Each visit method addresses the behaviour to be processed on the nodes of the AST. Both AST (one business model) and instance of the generated AST (an instance of the business model) are described in XML. In order to provide an easier way to

[35] To notify the flexibility and the rapidity of the development with SmartTools : the first operational prototype was achieved in three months.

input related information, it is possible to use the language *cosynt* that enables to define a concrete syntax (textual or graphical) with one or the other AST.

In order to preserve the flexibility of the prototype (maintenance, evolution, etc.), we use different visitors (at this time, thirteen). In fact, one visitor is created for each task to be performed; each of them could be replaced or removed according to the evolution of the prototype. It is not interesting to list all of them but they are classified in three main generation tasks: the Java classes related to the business model, those related to applications and other to data-model representations (AST and DTD). For example, according to the Java classes related to the business model, there are about seven instances of the design pattern Visitor (definition of hypergeneric parameters and characteristics, specification of actions, description of assertions, reification of both generic and non generic atoms, description of concepts).
Now, let us describe the output of those generators. The classes which are generated rely on built-in sub-hierarchies and on one MOP which includes facilities for *i)* handling the set of instances of an atom (its extension), taking into account the polymorphism, ii) loading/serializing instances of the model from/into XML files, iii) traversing instances of business models. Some of the sub-hierarchies encapsulate mechanisms for handling assertions, actions, aspects and facets. Others provide core hierarchies dedicated to the specification of business model meta-information and to the integration of both reification and meta-levels with the MOP.
Thanks to a small module, integrated within the MOP which enables the bootstrap of the main mechanisms (facet, application, etc.), we have started to describe (with SmartModels), business models dedicated to the description of aspect, facet and application (description of components will come next). This illustrates the self-extensible capability of our approach.

6 Related Work

Our work stands between the model approach and the AOP dedicated to DSLs, but in a broader context as we use the concept of software factory. This latter includes other notions for the design of applications such as software component for distributed applications.

The AOP-related works try to propose powerful mechanisms to describe the semantics of DSLs [3]. All of these works [19,25,27,18] stem from the basic issue of a better separation between the data structure and the semantics treatments.

It is well-known that the handling of an AOP can be rather complex and can introduce scarcely controllable situations [4]. To solve this problem, aspect-oriented languages dedicated to the context are proposed [29]. However, nearly in all the cases, the reflexivity mechanism plays major role [21,20]. Because of this, from our point of view, there is a strong dependence between the approach and the implementation techniques (that should be as less visible as possible at the model level).

On this point, our approach differs as generative programming from the model is used. We investigate how to directly introduce the aspect notion into the model. Indeed, one way to handle an AOP is to intentionally reduce the number of pointcuts. With the source code generation, our specialized AOP does not need reflexivity mechanisms that *i)* are inefficient and *ii)* impose a strong dependence on the semantics of the target language; the reflexivity mechanisms are different according to the implementation language

In comparison with approaches focused on the issues of modularity or reuse of semantics components [3,30], our approach is more oriented toward a family of DSLs. Other approaches investigate how to introduce powerful mechanisms to reuse language components; their objective is to be able to design a DSL by composition of existing components [3].

With regard to these approaches, we think it is important to propose a meta-level independent from any application and full of concepts. This meta-level enables the factorisation of the know-how related to a domain, used by different applications or facets. Additionally, as this meta-level is application-independent, it is less sensitive to structural changes and might be reused on another domain. This feature of reusing a facet (e.g. the display issue) on another domain seems very interesting.

In comparison with the modelling approaches, we are closer to those that advocate a domain model approach [1] than those that propose extensions (profiles) of a standard model. Indeed, having an universal model seems no longer be the solution advocating by the MDA but rather an approach à la MDA. From our point of view, the approaches such as *Action Semantics* [31] seem to have the same drawbacks i.e. to unify in the same framework (the UML notation) opposite notions.

Furthermore, our approach proposes simple mechanisms to describe treatments (e.g. the customisation of the entities, or the skeleton of the actions or facets) based on the structure of the semantics description. The semantics itself can be specified with a programming language or another formalism. We prefer centring our proposition around a meta-level (the concepts) and introducing generic mechanisms to enable the factorisation of basic concepts.

With respect to model transformation approaches (MDA) [1], our generation mechanisms are much more complex than those offered by simple transformations between models. A transformation at model level [5,1] will never performed complex operations, from our point of view, as models are semantically poor. But foreseeing a translation of the treatments from the original model to the target model [24] is important. On the one hand, the separation between the meta-level and the structural level, and on the other hand, the treatment specification with facets and aspects are certainly means to ease the translation or the reusability. This latter point needs to be addressed in our approach.

Finally, our approach, such as the one advocated in [12,6,11], is integrated in a much more global context: the software factories. Our software factory, SmartTools, already treats many of the concepts given in [12].

7 Conclusion and Perspectives

The object-oriented approach does not provide all the solutions even if it represents a valuable basis for the description of further approaches. This remark can also be applied to OO language extensions dealing, for example, with the separation of concerns. In particular, they do not provide a correct answer to the continuous evolution of the technologies: keeping applications up-to-date according to the evolution of technologies is too much time-consuming. We believe that to provide an approach centered on models which capture the know-how, independently from both the software platform and the possible applications is promising.
In this paper, we propose to structure the framework of Model-Oriented Programming with a set of essential rules. We consider them as a first attempt for the definition of the main principles of this approach. A second contribution is the proposal of SmartModels which is one interpretation of those rules. Finally, we demonstrate the feasibility of the approach with a prototype (SmartFactory), which thanks to the use of SmartTools may apply to itself a model-oriented approach.

Our perspectives are twofold. Firstly we want to experiment our approach for the description of various business models and their applications; currently we start to investigate the business model of a the French electricity company, EDF. The objective is to get feedbacks in order to improve the expressiveness of SmartModels as well as a better automation (in SmartFactory) of *i)* the generation of the behaviour, and *ii)* the semantics transformation of both business models and applications when they evolve toward another model or application.

Secondly, we want to improve the expressiveness of the business models for the description of facets, aspects, applications and components, and then to implement them with SmartModels. Through the definition of those business models which are dedicated to enrich SmartModels itself, we aim to improve the quality and the percentage of code automatically generated.

References

1. Colin Atkinson and Thomas Kühne. The role of meta-modeling in MDA. In Jean Bezivin and Robert France, editors, *Workshop in Software Model Engineering*, 2002.
2. Isabelle Attali, Carine Courbis, Pascal Degenne, Alexandre Fau, JoÃ«l Fillon, Didier Parigot, Claude Pasquier, and Claudio Sacerdoti Coen. SmartTools: a development environment generator based on XML technologies. In *XML Technologies and Software Engineering*, Toronto, Canada, May 2001. ICSE'2001, ICSE workshop proceedings. ftp://ftp-sop.inria.fr/oasis/personnel/Carine.Courbis/smarticse02.pdf.
3. Don Batory, Bernie Lofaso, and Yannis Smaragdakis. JTS: Tools for implementing Domain-Specific Languages. In P. Devanbu and J. Poulin, editors, *Proceedings: Fifth International Conference on Software Reuse*, pages 143–153. IEEE Computer Society Press, 1998.

4. Noury M. N. Bouraqadi-Saâdani and Thomas Ledoux. Le point sur la programmation par aspects. In *Technique et Sciences Informatiques*, volume 20, page 505 à 528. HermÃ¨s, 2001.
5. Jean BÃ©zivin. From Object Composition to Model Transformation with MDA. In *TOOLS USA*, Santa-Barbara, August 2001. IEEE TOOLS-39.
6. Steve Cook and Stuart Kent. The Tool Factory. In *OOPSLA'2003, workshop on Generative Techniques in the context of MDA*, Anaheim - USA, October 2003.
7. Pierre Crescenzo and Philippe Lahire. Using both specialisation and generalisation in a programming language: Why and how? *Lecture Notes in Computer Science*, 2426:64–73, 2002.
8. Krysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Techniques, and Applications*. Addison-Wesley, June 2000.
9. Krzysztof Czarnecki and John Vlissides. Domain-Driven Development. Special Track at OOPSLA'03 URL: http://oopsla.acm.org/oopsla2003/files/ddd.html.
10. P. Desfray. *Object Engineering, the Fourth Dimension*. Addison-Wesley Publishing Co., 1994.
11. Christer Fernström, Kjell-Håkan Närfelt, and Lennart Ohlsson. Software factory principles, architecture, and experiments. *IEEE Software*, 9:36–44, March 1992.
12. Jack Greenfield and Keith 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.
13. Object Management Group. Meta Object Facility (MOF) specification (version 1.3). Technical report, Object Management Group, March 2000.
14. OMG Staff Strategy Group and Richard Soley. Model-Driven Architecture. Technical report, OMG, November 2000.
15. 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.
16. 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.
17. 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.
18. Karl J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996. ISBN 0-534-94602-X.
19. Karl J. Lieberherr and Doug Orleans. Preventive program maintenance in Demeter/Java. In *Proceedings of the 1997 International Conference on Software Engineering*, pages 604–605. ACM Press, 1997.
20. Cristina Videira Lopes and Karl J. Lieberherr. AP/S++: A CASE-study of a MOP for purposes of software evolution. Technical Report NU-CCS-95-?, Xerox PARC and Northeastern University, November 1995.

21. Jacques Malenfant and Pierre Cointe. Aspect-Oriented Programming versus Reflection: a first draft. In *Position Statement for the OOPLSA '96 AOP meeting*, 1996.
22. Dino Mandrioli and Bertrand Meyer, editors. *Advances in Object-Oriented Software Enineering*. Prentice Hall, New York, 1992.
23. Todd Millstein and Craig Chambers. Modular statically typed multimethods. In R. Guerraoui, editor, *Proceedings ECOOP'99*, LCNS 1628, pages 279–303, Lisbon, Portugal, June 1999. Springer-Verlag.
24. OMG. MDA - Model-Driven Architecture. http://www.omg.org/mda.
25. Doug Orleans and Karl Lieberherr. DJ: Dynamic adaptive programming in Java. *Lecture Notes in Computer Science*, 2192:73–??, 2001.
26. Jens Palsberg and C. Barry Jay. The essence of the visitor pattern. In *COMPSAC'98, 22nd Annual International Computer Software and Applications Conference*, Vienna, Austria, August 1998.
27. Jens Palsberg, Boaz Patt-Shamir, and Karl Lieberherr. A new approach to compiling adaptive programs. *Science of Computer Programming*, 29(3):303–326, 1997.
28. Charles Simonyi. The death of programming languages, the birth of intentional programming. Technical report, Microsoft, Inc., September 1995.
29. Yannis Smaragdakis and Don Batory. DiSTiL: A transformation library for data structures. In *USENIX Conference on Domain-Specific Languages*, 1997.
30. Y. V. Srinivas and Richard Jullig. Specware(TM): Formal support for composing software. Technical Report KES.U.94.5, Kestrel Institute, 1994. see also Proceedings of the Conference on Mathematics of Program Construction, Kloster Irsee, Germany.
31. Gerson Sunyé, François Pennaneac'h, Wai-Ming Ho, Alain Le Guennec, and Jean-Marc 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 *LNCS*, pages 433–447. Springer, 2001.
32. Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 1998.
33. T Ziadi, B Traverson, and Jean-Marc JÃ©zÃ©quel. From a UML Platform Independent Component Model to Platform Specific Component Models. In *International workshop in Software Model Engineering (WiSME02) at UML2002*, Dresden (Germany), September 2002.