SMARTFACTORY - an Implementation of the Domain Driven Development Approach

Emanuel Țundrea*, Philippe Lahire, Didier Parigot***, Ciprian-Bogdan Chirilă*, Dan Pescaru***

* "Politehnica" University of Timişoara - Faculty of Automatics and Computer Science - V. Pârvan 2, 300223, Timişoara, România
Emanuel@emanuel.ro, Chirila@cs.utt.ro, Dan@cs.utt.ro
http://www.cs.utt.ro

** Laboratoire I3S (UNSA/CNRS) - Projet OCL - 2000 route des Lucioles - Les Algorithmes, Bâtiment Euclide - BP 121 - F-06903, Sophia-Antipolis cedex - France
Philippe.Lahire@unice.fr
http://www.i3s.unice.fr

*** INRIA Sophia-Antipolis - 2004, route des Lucioles - BP 93 - F-06902 Sophia-Antipolis cedex – France
Didier.Parigot@inria.fr
http://www.inria.fr

Abstract. *In the world of software everything evolves. So, then, do systems engineering on software architectures, object oriented languages and technologies. The object-oriented programming (OOP) principles did not cure important issues faced by software companies these days on developing complex software for reuse and protecting the more and more evolving applications against technological obsolescence. In this direction there are a lot of approaches like Aspect Oriented Programming, Subject Oriented Programming, Intentional Programming, Web Services, generative techniques and component technology. Each one of them tries to solve a problem from OOP. In this paper we want to propose our approach called SMARTFACTORY which is based on the vision of OMG: Domain-Driven Development. It is a modelling support for software factories centered on models, technological neutral and which introduces a new meta-level on top of the classical programming entities. We will present i) concepts we use for handling the meta-information of a business-model (SMARTMODELS), ii) how we describe its entities, iii) how to generate and adapt more effective the application building process and iv) we will also address at each level implementation issues of the prototype (SMARTFACTORY) which benefits from earlier experiences.*

Keywords: software engineering, business model, Domain-Driven Development (DDD), Aspect-Oriented Programming (AOP), Subject Oriented-Programming (SOP), Generative Programming

1 Introduction

All the trends in software engineering today target the idea of developing software more efficient through reuse. In this direction there are a lot of approaches like AOP (*Aspect-Oriented Programming*) [7], SOP (*Subject-Oriented Programming*) [5], IP (*Intentional Programming*) [13], Web Services [15], generative techniques [3] and component technology [14]. Each one of them answers a couple of issues where general principles of the object-oriented programming did not cure important problems.

A very promising approach towards a better and more practical framework for software development seems to be the vision of OMG: Model-Driven Architecture [10] and Domain-Driven Development [4]. They propose to separate platform-independent business-models (PIM) from the perspective of platform-specific design and implementation models (PSM). Taking into consideration the new MDA paradigm, generative programming and other trends in current research communities for software engineering we propose a new and challenging view on developing software: *Model-Oriented Programming* [8].

2 SMARTMODELS: an Attempt to Introduce Model-Oriented Programming

SMARTMODELS is a set of domain specific models dedicated to the development of software. It introduces also the new paradigm Model-Oriented Programming [8] and it supports a practical interpretation of its principles and rules for software development engineering (SMARTFACTORY). SMARTMODELS is an original approach because: i) it integrates new ideas from AOP [7], SOP [5], IP [13], software agents [15] and design patterns to build flexible, readable and easy maintainable software; ii) it is business-model based which provides the possibility to encapsulate the specific knowledge of a domain according to multi-system scope development found in domain engineering [4]; iii) it proposes a way to handle the treatment of a business-model semantics at the level of its generic entities and at the level of its derived entities (this will also affect their behavior); iv) it ensures a clear separation between the model and the technologies which makes the model executable by a software platform, but it has a solid fundament in order to map to any platform; v) it is a modeling support for the design of software factories which rely on a data-model (SMARTFACTORY) which automate as much as possible the code generation, provides easy and clear entry points in the generated source-code for the user to change or update it and also it relies on OMG and W3C standards like XML and DTD for serialization of the models, OCL for the assertions, MOF and AST for the meta-model description.

2.1 The Framework: SMARTTOOLS

The implementation of the SMARTMODELS approach, the SMARTFACTORY project, was developed in the framework of SMARTTOOLS [1] and DDD [4]. It is a development environment generator that provides a structure editor and semantic tools as main features. It was built on Java and XML technologies. Therefore it offers support for easy designing of new software development environments for programming languages as well as domain specific languages defined with XML.

In order to be able to define a new language SMARTTOOLS uses its own abstract and independent defined formalism called Abstract Syntax Tree (AST) – "Absynt" which is close from the BNF. To be compatible with the standard formalisms SMARTTOOLS incorporates a couple of translators. In this way a meta-model of a business-model can be described also as a DTD. When an AST is imported SMARTTOOLS automatically generates: i) a *set of Java classes* that extend a DOM implementation for each operator of the language; ii) a *parser for the XML documents* (the source-code of the new language). When this file is parsed SMARTTOOLS creates an instance for each node according to the classes generated at the DTD importation and tests the conformity of this tree structure of instances with the DTD. Also, during editing the application, the environment guarantees that the XML files remain valid and well-formed; and iii) a *default visitor class* which offers the possibility to traverse through the DOM tree structure of nodes of the application. According to the different concerns a user can have, he may extend this visitor class through inheritance and override any visit methods.

Following this specification we defined our description (meta-model) of a business-model as an AST. The root of our formalism is:

```
Top = model(GenInfo? genInfo, InheritModel? inheritModel, Concept[]
concepts, Atom[] atoms, DerivedAtom[] derivedAtoms,DerivedConcept[]
derivedConcepts, CustomizableRelationshipsDefinition[]
RelationshipsDefinitions, ModifierDefinition[] modifierDefinitions,
BasicTypeDefinition[] basicTypeDefinition);
```

This description points out the various entities of a business-model and each one of them will be explained in the next sections continuing to address implementation issues (SMARTFACTORY) from the perspective of our approach (SMARTMODELS). We specialized the default SMARTTOOLS visitor in order to build generators for each entity from our approach (the current output produces Java classes).

2.2 The Starting Point: Meta-Object Protocol

The kernel of SMARTFACTORY is represented by a Meta-Object Protocol (MOP) and the abstract reification of the entities of our meta-model. MOP consists of a set of Java classes which encapsulates the basic principles and the definitions of the entity types of our approach [8].

Its architecture was driven by previous works on OFL [2] and it is formed by: i) a class (in fact the so-called MOP) which retains the place in the structure of nodes of each entity of a business-model (its name, the name of the super-class specialized by this entity and the name of the meta-class – the entity that provides the meta-information) and its extent (the collection of its instances). This collection is updated automatically when a new instance is created and through the previous information it is possible to implement a management on the list of instances of entities; ii) a set of classes which provides basic tools for managing the entities: collection, OCL basic types (enumeration, tuple, integer, boolean, etc.) [11], OFL values, types of redefinitions for parameters and characteristics (see section 2.3) and OFL constants; iii) a set of classes which define the abstract core representation of each entity of a business-model.

The kernel is built-in and consists of a set of entities. This is a set of classes (currently Java), which are manually created. Yet, as a consequence of the fact that any entity is a "first-class" entity (they are modeled through the specification of a class.This means that it may be handled using the full expressiveness of the implementation language - i.e. specialized through inheritance), the kernel can be bootstrapped. It represents an open and flexible platform to describe business-models.

A model is defined through the identification of its entities (see next sections) according to the know-how of a specific domain[1]. This process consist in producing an XML document (i.e. by a parser of the domain specific language) compliant with the AST (or DTD) which describes a model in our approach. This document will drive the generation process (making the business model executable with SMARTFACTORY) of a class (currently Java) for each entity. The resultant implementation of the model is attached to MOP as sub-hierarchies of the built-in kernel (adding features for handling access to the specialized and meta hierarchies, to its extension, for loading/saving instances of entities from/into XML streams).

In the next sections we will present each entity of SMARTMODELS with respect to the level it manifests. Figure 1 distinguishes between the different levels of the architecture of our meta-model in order to define business-models. They will be used by generators in SMARTFACTORY to produce code attached to the MOP.

2.3 The Meta-Level: Concepts for Handling Meta-Information

In this section we want to present the entities which participate to the definition and the management of the meta-information of a business-model. First of all, a concept encapsulates the semantics of entities and their treatments. It can be related to one or a number of atoms (see section 2.4) and drives their behavior. A concept makes the clear distinction between the semantics (meta-level) of the

[1] This approach follows the Domain-Driven Development [4] principles and therefore offers a framework for the development of domain-specific applications.

entities of a business-model and their reification. As positive consequences: i) the maintenance of the semantics (updating and redefining of the semantics) deals only with concepts; ii) the support for reuse of the semantics in other (closely related) models; iii) the model transformation (one of the key points of our approach).

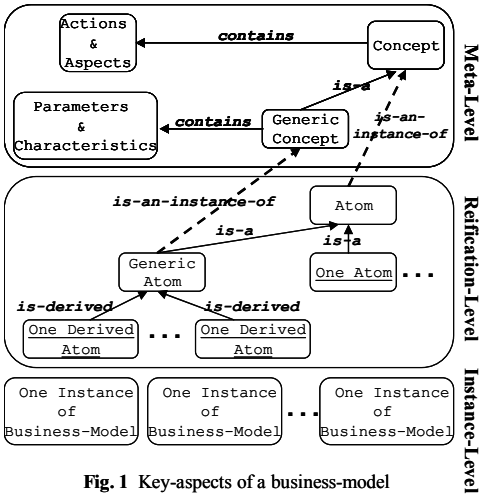

Fig. 1 Key-aspects of a business-model

At this point it is important to mention that the elements that compound the concepts address only the semantics of the entities and not their instances The semantics of a business-model is reified through a set of hypergeneric parameters and characteristics [2] (which form the meta-information) and a set of actions (which perform treatments on the entities according to their meta-information) which describes a part of the model semantics[2]. The identification of the parameters and characteristics and their possible values is the job of the meta-programmer which addresses the know-how of the business-domain.

The hypergeneric parameters customize the behavior of the entities[3] of a business-model. Their role is to capture and express the properties which compound the definition of the generic entities. A parameter expresses a basic type property, e.g. a boolean or an integer, an enumeration, a tuple or a collection of values. A characteristic expresses a property whose value is defined by an atom or a set of atoms . In order to describe the behavior of a generic entity the programmer has to set those values. For example, a business-model built to encapsulate the structures (entities) and semantics of an OOP language may define *parameters* like: *cardinality* which expresses if there is simple or multiple inheritance, *generator* which specifies if the given entity can create or not its own instances; or *characteristics* like: the collection of valid kinds of classifiers for a given type of inheritance.

Actions are "first-class" entities addressed by concepts in order to dynamically manage the behavior of atoms according to their meta-information. The body of an action encapsulates the execution which can be performed by that action. It depends on: i) querying the parameters and characteristics of the generic atom to

[2] At the moment the body of an action corresponds to some declarative information which allows the generator to make easier the job of the user but the action has to be described manually in Java. A dedicated pseudo language is undergoing definition.

[3] It refers to generic atoms (see section 2.4) and not their instances

which is attached the action; ii) a set of invariants, preconditions and postconditions; iii) an optional set of aspects (see section 2.5); iv) additional information provided by the meta-programmer.

Thanks to AOP paradigm, it is also possible to insert new concerns with respect to model' semantics. This is completely independent from the category of visit-entities from the potential applications (see section 2.5) and they were implemented in order to easy add new pieces of behavior which are orthogonal to the semantics.

2.4 The Reification-Level: Atoms for Describing Entities

We arrived at the line of demarcation between semantics and data of a business-model. As we anticipated, an atom is the reification of entities of a business-model. The task of a programmer is to identify the entities of a specific domain conform to each level of the model it describes. Through the generation process with SMARTFACTORY this set of classes will be attached to its kernel (chapter 3). The atoms can be organized in hierarchies through inheritance (same as concepts). Thus we provide a framework for ease model transformation and increment.

Although not all atoms use this facility, each atom has its meta-information in the corresponding concept. An atom is seen as an instance of its concept (see figure 1). However, there are two axes of co-ordinates that we use to distinguish between atoms: atoms which are generic or not. The genericity is a reflection of the semantic-level which specifies if the meta-information of a given entity has or not parameters and characteristics (see section 2.3). A business-model designer may define entities which need semantic information (which becomes part of an atom definition) and we call them "generic atoms". There are atoms which do not need additional customization besides their reification (their behavior does not depend on parameters) and we call them "atoms without parameters"; and ii) atoms which have instances within applications or not. The generic atoms may or may not have instances at the application-level. MOF [9] makes this distinction through the notion of abstract class. We believe it is not sufficient because even if a class is just partially defined we can not say that no applications will need it whatever it is the context of use. That is why SMARTMODELS choused the notion of *derived atom* (see figure 1) which is an instance of a generic atom obtained through relevant combination of values associated with the sets of characteristics and parameters which participate to the definition of its generic atom. This meta-information makes more precise the role of an atom in the application building process and its behavior there and it is even more interesting if it is imported in other models.

Let's take again an example of an object-oriented language. Illustrations of *atoms without parameters* may be attribute, method, method parameters, modifiers. A generic atom defining a classifier may distinguish between different derived atoms like: abstract or concrete class, inner class, interface. If we glance at Java language possible relationships we may find *extends* between classes, *extends* between

interfaces or *implements* between an interface and a class, all of them defined as derived atoms of a generic "relationship between classes" atom. All this properties are recorded in their meta-level through parameters (see section 2.3).

2.5 The Application-Level: Facets for Implementing Concerns

Applying appropriately the SMARTMODELS principles (see section 2.3/2.4) should lead to a much more effective application building process with SMARTFACTORY. In our approach this process consists of a set of traversals of the graph of atoms corresponding to a business-model. A type of traversal is the main entity which influence upon the way an application must be developed and we call it *facet*. The organization by facets of an application draws from SOP [5] and ASoC [7].

A facet represents one concern of the application with respect to the business-model. Because of that and because our models may be described as *abstract syntax trees* (ASTs) we based our approach on the Visitor pattern[4] [12]. Therefore a facet is mainly compound by a set of *visit-entities*. A visit-entity implements a treatment on an entity of the business-model. It is described by an execute method which contains not only the behavior of the visit, but also a mechanism for the description and the verification of preconditions and postconditions. It has a mechanism for binding the business-model and has direct access to the properties of the entity it handles and to the context of the execution. Also, a facet specifies: i) the business model that it addresses; ii) how the business model is visited (the type of the traversal); iii) the entities that are relevant according to the objectives, and possibly iv) additional technologies that it needs.

Technologies are defined independently from SMARTMODELS (e.g., by an API or a library of classes). They contain functionalities which allow to define more easily the application (e.g. DOM API is welcomed to manipulate XML representations of the business models). The source code generation is essential to our approach because it allows the programmer, to focus only on the visit-entities that are addressed by the facet and also to be assisted for the description of their behavior.

Organizing facets by visit-entities offers a couple of very important advantages: i) it is a very flexible framework which allows easily transformation of the behavior of the applications; ii) it is dependent on the structure of the model and it can traverse it at different levels of granularity; iii) it has the possibility to carry information during the traversal either up (toward the root of the facet – by default the starting atom is the root atom of the model) or down (toward the dispatched visit-entities); and iv) it handles orthogonal concerns and assertions.

---

[4] Visitors are particularly very attractive in program-analysis and CASE applications. In our approach we used SMARTTOOLS [1] framework and research on applying the visitor pattern technique for tree manipulations.

We classify the facets according to the type of traversal. For now we have three deep-first traversals, but more kinds will be proposed depending on the needs of the programmers: i) *plain facet*: it consist of a set of visit-entities each one of them corresponding to only one entity of the model. The possibility to explore the values of all the visible properties of the entity and to control the navigation policy are open capabilities and the programmer can decide what to query and to redirect the traversal; ii) *detail facet*: it is similar to the previous type, but the granularity is enhanced as a visit-entity corresponds to a property of an entity. In this way the detail facet will contain as many visit-entities as many properties have the target entities; and iii) *hide facet*: it hides the traversal and thus the programmer can not control the traversal. The use of hide facet is less complex and it favors the reuse of the application semantics of atoms who evolve. It also can be plain or detailed.

Invariants, pre/postconditions are first-class entities and like in programming languages such as Eiffel, they can be evaluated at the beginning and at the end of a method; they determine whether the visit of one atom succeeded or failed. We are now investigating how to describe them in SMARTMODELS using the OCL [11].

Aspects for Applications. An aspect can be absolutely free attached to several visit-entities or to several facets (it will be applied to all their visit-entities) of a given application. They will be performed at some points of the execution of one or several visit-entities, one or several facets (for example, to check the validity of a constraint, to load data, to check access rights or to trace a method call).

3 SMARTFACTORY – An Implementation Centered on Models

In section 2.1 we introduced the SMARTFACTORY prototype in the framework of SMARTTOOLS and in sections 2.3 and 2.4 we presented how to describe a business-model with our approach SMARTMODELS continuing to point out implementation issues. In the previous sections we concentrated more on how to develop with SMARTFACTORY and how the programmer will use it. In this section we want to insist some more on a couple of important practical elements on how SMARTFACTORY was developed with SMARTTOOLS.

In order to preserve the flexibility of our prototype (maintenance, evolution, etc.), we adapted different SMARTTOOLS visitors (at this time, fourteen). In fact, one visitor (generator) is created for each task to be performed; each of them can be replaced or removed according to the evolution of the prototype. It is not interesting to list all of them, but they may be classified in three main generation tasks: i) one part of the generators builds the entities of a business-model (the meta-information and the reification), ii) another part generates different model representations (AST/DTD) iii) and some automate the application generation

process (providing an application development framework: generators for the domain of the business-model). 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, assertions, reification of generic and non generic atoms, description of concepts).

The output of those generators are classes which 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 derived atoms will come next). This illustrates the self extensible capability of our approach.

Conclusions and Perspectives

In the world of software everything evolves. So, then, do systems engineering on software architectures, object oriented languages and technologies. It has to respond to the new user needs for more and more complex software and for protection against technological obsolescence and platforms rapid evolution. The object-oriented programming principles did not cure important issues faced by software companies these days and this remark can also be applied to the modeling techniques defined in the last decade. In this paper we propose an implementation of the DDD [4] principles called SMARTFACTORY. It represents the first practical validation of our approach centered on models called SMARTMODELS and we intended to present its key-aspects. In order to get a better understanding of the meta-model, at every section we defined the concepts we use for handling the meta-information of a business-model (SMARTMODELS), the description of its entities and aspects on how to generate and adapt more effective the application building process. We also addressed at each level implementation issues on SMARTFACTORY developed in the framework of SMARTTOOLS.

Our perspectives are twofold. Firstly we want to experiment our approach for the description of various business models and their applications. The objective is to get feedbacks in order to improve the expressiveness of SMARTMODELS as well as a better automation (in SMARTFACTORY) of the generation of the behaviour, and of 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, derived atoms and applications, and then to implement them with SMART-

FACTORY. Through the definition of those business models which are dedicated to enrich the meta-model itself, we aim to improve the quality and the percentage of code automatically generated.

References

[1] Isabelle Attali, Carine Courbis, Pascal Degenne, Alexandre Fau, Didier Parigot, Claude Pasquier and Claudio Sacerdoti (May 2001), SMARTTOOLS: a development environment generator based on XML technologies, *XML Technologies and Software Engineering*, Toronto, Canada, ICSE'2001 workshop.

[2] Pierre Crescenzo, Philippe Lahire (2002), Using both specialisation and generalisation in a programming language: Why and how?, vol. 2426, *Lecture Notes in Computer Science*, pages 64-73.

[3] Krysztof Czarnecki and Ulrich W. Eisenecker (June 2000), Generative Programming: Methods, Techniques, and Applications, Addison-Wesley.

[4] Krzysztof Czarnecki and John Vlissides (2003), Domain-Driven Deve-lopment, Special Track *OOPSLA'03*, `http://oopsla.acm.org/ddd.html`

[5] William Harrison, Harold Ossher (October 1993), Subject-Oriented Programming – A critique of pure objects, *Proceedings ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, ACM Press, pages 411-428.

[6] George Heineman, William T. Councill (May 2001), Component-Based Software Engineering – Putting the Pieces Together, Addison-Wesley.

[7] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Marc Loingtier, John Irwin (June 1997), Aspect-Oriented Programming, *ECOOP'97 – Object-Oriented Programming 11th European Conference*, Jyväskylä, Finland, vol 1241, *Lecture Notes in Computer Science*, pages 220-242, Springer-Verlag.

[8] Philippe Lahire, Didier Parigot, Carine Courbis, Pierre Crescenzo, Emanuel Ţundrea, *An Attempt to Set the Framework of Model-Oriented Programming*, Buletinul Ştiinţific al Universităţii „Politehnica" din Timişoara, România, Seria Automatică şi Calculatoare, Periodica Politechnica, Transactions on Automatic Control and Computer Science, Vol. 49 (63), 2004, ISSN 1224-600X.

[9] Object Management Group (March 2000), Meta Object Facility (MOF) Specification, Version 1.3, Technical Report, OMG.

[10] Object Management Group, MDA, `http://www.omg.org/mda`

[11] Object Management Group (January 6, 2003), Object Constraint Language, RfP document ad/2000-09-03, Version 1.6, OMG.

[12] Jens Palsberg, Barry Jay, The Essence of the Visitor Pattern, *COMPSAC'98, 22nd Annual International Computer Software and Applications Conference*, Vienna, Austria.

[13] Charles Simonyi (September 1995), The Death of Programming Lan-guages, the Birth of Intentional Programming, Technical Report, Microsoft, Inc.

[14] Clemens Szyperski (1998), Component Software: Beyond OOP, ACM Press and Addison-Wesley.

[15] W3C Working Group (February 11, 2004), Web Services Architecture, `http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/wsa.pdf`