# Code Generation for a Bi-dimensional Composition Mechanism

Jacky Estublier, Anca Daniela Ionita, Tam Nguyen

# Code Generation for a Bi-dimensional Composition Mechanism

Jacky Estublier[1], Anca Daniela Ionita[2], Tam Nguyen[1]

[1]LIG-IMAG, 220, rue de la Chimie BP5338041 Grenoble Cedex 9, France
{Jacky.Estublier, Tam.Nguyen}@imag.fr
[2] Automatic Control and Computers Faculty, Univ. "Politehnica" of Bucharest, Spl.Independentei 313, 060042, Bucharest, Romania Anca.Ionita @ aii.pub.ro

**Abstract. C**omposition mechanisms are intended to build a target system out of many independent units. The paper presents how the aspect technology may leverage the hierarchical composition, by supporting two orthogonal mechanisms (vertical and horizontal) for composing completely autonomous parts. The vertical mechanism is in charge of coordinating heterogeneous components, tools or services at a high level of abstraction, by hiding the technical details. The result of such a composition is called "domain" and, at its turn, it represents a high granularity unit of reuse. The horizontal mechanism composes domains at the level of their abstract concepts, even if they have been independently designed and implemented. The paper discusses the formalization of the vertical and horizontal compositions, and the wizard we have developed for generating the needed code (using Aspect Oriented Programming) in order to build the modeled applications.

**Keywords:** Model Driven Engineering, code generation, AOP, model composition, Domain Engineering

## 1 Introduction

Creating software based on already available components is an obvious way to speed up the development process and to increase productivity. The "classical" composition approach - often referred as CBSE (Component Based Software Engineering) – deals with components especially designed to be composed and with a hidden internal structure. This approach works well under constraints related to context dependency and component homogeneity. These constraints involve a rigid composition mechanism, since components know each other, must have compatible interfaces and must comply with the constraints of the same component model, which reduces the likelihood of reuse and prevents from obtaining a large variety of assemblies.

The paper presents an alternative composition approach, which still sticks to the encapsulation principle (parts have a hidden internal structure) and reuse components without any change, but which relaxes the composition constraints found in CBSE. The aims of this approach can be summarized as below:

A. The components or, generally speaking, the parts ignore each other and may have been designed and developed independently, i.e. they do not call each other;

B. Composed parts may be of any nature (ad hoc, legacy, commercial, COTS, local or distant);
C. Parts are heterogeneous i.e. they do not need to follow a particular model (component model, service etc.);
D. Parts have to be reused without performing any change on their code.

To solve the heterogeneity issue (the above mentioned aims B and C), one can imagine that the part to be composed is wrapped, directly or indirectly, into a "composable element" [1]. For composing parts that ignore each other and have been designed independently (aim A), there is a need to define a composition mechanism that is not based on the traditional method call.

The publish subscribe mechanism [2] is an interesting candidate, since the component that sends events ignores who (if any) is interested in that event, but the receiver knows and must declare what it is interested in. If other events, in other topics are sent, the receiver code has to be changed. Moreover, the approach works fine only if the sender is an active component. Aspect Oriented Software Development (AOSD) [3][4] satisfies some of the requirements above, since the sender (the main program) ignores and does not call the receiver (the aspects). Unfortunately, the aspect knows the internals of the main program, which defeats the encapsulation principle [5] and aspects are defined at a low level of abstraction (the code) [6][7].

The bi-dimensional composition mechanism presented here is intended to be a solution for such situations. The idea is that the elements to be composed are not traditional components, but much larger elements, called domains, which do not expose simple interfaces, but (domain) models (described in chapter 2). Composition is not performed calling component interfaces, but composing such (domain) models. Model composition allows the definition of variability points, which make the mechanism more flexible than component composition [8]. In contrast with a method call, model composition does not require from the models to stick to common interfaces, or to know each other, it may even compose independent concepts.

One of our main goals is also to reuse code without changing it (aim D) because:

- we have to compose tools, for which we do not have access to the internal code, but to an API only - this is what we call vertical composition and results in a so-called domain (see chapter 2.1 about the concepts and chapter 3 about the code generation);
- we want to reuse domains (which may be quite large) without changing them, because any change would require new tests and validations. For this purpose, we apply the horizontal composition (described in chapter 2.2. and 4).

So, our method is non-invasive, using an implementation based on AOP; the composed domains and their models are totally unchanged and the new code is isolated with the help of aspects. However, since the AOP technique is at code level, performing domain composition has proved to be very difficult in practice; the conceptual complexity is increased, due to the necessity to deal with many technical details. The solution would be to specify composition at a high, conceptual level and to be able to generate the code based on aspects.

The elevation of crosscutting modeling concerns to first-class constructs has been done in [9], by generating weavers from domain specific descriptions, using ECL, an extension of OCL (Object Constraint Language). Another weaver constructed with domain modeling concepts is presented in [10], while [11] discusses mappings from a design-level language, Theme/UML, to an implementation-level language, AspectJ.

For managing the complexity in a user friendly manner, we propose a conceptual framework, such as the AOP code is generated and the user defines the composition at a conceptual level, using wizards for selecting among pre-defined properties, instead of writing a specification in a textual language. Mélusine is the engineering environment that assists designers and programmers for developing such autonomous domains, for composing them and for creating applications based on them [1]. Recently, Mélusine has been leveraged by a new tool, which supports domain composition by generating Java and AspectJ code; it guides the domain expert for performing the composition at the conceptual level, as opposed to the programming level.

Chapters 3 and 4 describe the metamodels that allow code generation for vertical and horizontal composition. Chapter 5 compares the approach with respect to other related works and evaluates its usefulness when compared with the domain compositions we have performed before the availability of the code generation facility.

## 2 A Bi-dimensional Composition Technique

A possible answer to the requirements presented above is to create units of reuse that are *autonomous* (eliminating dependencies on the context of use) and *composable* at an *abstract level* (eliminating dependencies on the implementation techniques and details). The solution presented here combines two techniques (see Fig. 1):

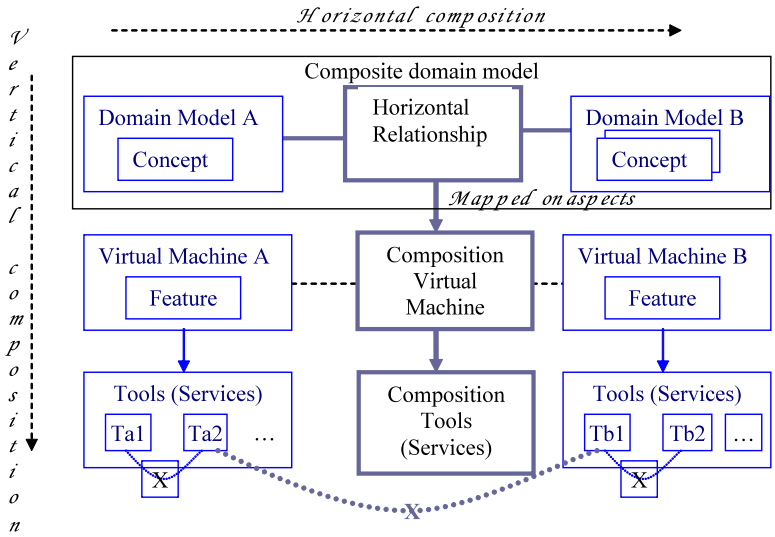


**Fig. 1.** Bi-dimensional composition mechanism

- *Building autonomous domains* using *vertical composition* - which is a coordination of heterogeneous and "low level" components or tools, in order to provide an homogeneous and autonomous functional unit, called domain;

- *Abstract composition of domains* using *horizontal composition* – performed between the abstract concepts of independent domains, without modifying their code.

### 2.1 Developing Autonomous Domains: Vertical Composition

Developing a domain can be performed following a top-down or a bottom-up approach. From a top down perspective, the required functionalities of the domain can be specified through a model, irrespective of its underlying technology; then, one identifies the software artifacts (available or not) that will be used to implement the expected functionality and make them interoperate. From a bottom up perspective, the designer already knows the software artifacts that may be used for the implementation and will have to interoperate; therefore, the designer has to identify the abstract concepts shared by these software artifacts and how they are supposed to be consistently managed; then, one defines how to coordinate the software artifacts, based on the behavior of the shared concepts.

In both cases, the composition is called vertical, because the real software components, services or tools are driven based on a high level model of the application The model elements are instances of the shared concepts, which are abstractions of the actual software artifacts. The synchronization between these software artifacts and the model means that the evolution of the model is transformed into actions performed by the software artifacts.

The set of the shared concepts and their consistency constraints constitute a domain model, to which the application model must conform to. In the Model Driven Engineering (MDE) vocabulary, the domain model is the metamodel of all the application models for that domain [6]. For instance, one of our domains, which has been intensely reused, is the *Product* domain, which will also be presented in the case study of this paper. It was developed as a basic versioning system for various products, characterised by a map of attributes, according to their type; the versions are stored in a tree, consisting of branches and revisions. The domain model of *Product* domain contains the following concepts: Product, Branch, Revision, Attribute, ProductType, ProductAttribute, AttributeType (see [8] for a detailed presentation).

The application models are interpreted by a virtual machine built according to the domain model, which orchestrates then the lower level services or tools (see Fig. 1). The domain interpreter is realized by Java classes that reify the shared concepts (the domain model) and whose methods implement the behavior of these concepts. In many cases, these methods are empty because most, if not all, the behavior is actually delegated to other software artifacts, with the help of aspect technology. Thus, the domain interpreter, also called the domain virtual machine, separates the abstract, conceptual part from the implementation, creating architecture with 3 layers [6] (see Fig. 1). The domains may be autonomously executed, they do not have dependencies and they may be easily used for developing applications (details in chapter 3).

For the example of *Product* domain, one of our application models is dedicated to the J2EE architecture, versioning typed software artefacts. A Servlet from this application model conforms to the concept ProductType from the *Product* domain model. Moreover, in order to assure its persistency, the *Product* domain interpreter may use one of the domain tools, based either on SQL storage, or on the repository of another

versioning system, like Subversion or CVS. They correspond to the third layer presented in Fig. 1. The tool is then chosen with respect to the client preference.

### 2.2 Abstract Composition of Domains: Horizontal Composition

It may happen that the development of a new application requires the cooperation of two concepts, pertaining to two different domains, and realized through two or more software components, services or tools. In this case, the interoperation is performed through a horizontal composition between these abstract concepts, and also through the domain virtual machines, ignoring the low level components, services, tools used for the implementation. The mechanism consists in establishing relationships between concepts of the two domain models and implementing them using aspect technology, such as to keep the composed domains unchanged. A very strict definition of the horizontal relationship properties is necessary, such as to be able to generate most of the AOP code for implementing them. This code belongs to the Composition Virtual Machine (Fig. 1) and is separated from the virtual machines of the composed domains. This composition is called horizontal, because it is performed between parts situated at the same level of abstraction. It can be seen as a grey box approach, taking into account that the only visible part of a domain is its domain model. It is a non-invasive composition technique, because the components and adapters are hidden and are reused as they are (details in chapter 4).

The composition result is a new domain model (Fig. 1) and therefore, a new domain, with its virtual machine, so that the process may be iterated. As the domains are executable and the composition is performed imperatively, its result is immediately executable, even if situated at a high level of abstraction.

Fig. **2** is a real example of two domain models –used and reused in our industrial applications. On the left, there is the *Activity* domain, which supports workflow execution, while on the right there is the *Product* domain, meant to store typed products and their attributes. The light colored boxes represent the visible concepts (the abstract syntax) used for defining the models with appropriate editors; the dark grey ones show the hidden classes, introduced for implementing the interpreters (the virtual machines).

For each domain, a model is made by instantiating the concepts from the light colored part. Fig. 3 shows an Activity model, conforming to the metamodel from Fig. 2; the boxes for *Design*, *Programming* and *Test* are instances of the *ActivityDefinition* concept; connector labels, like *requirement*, *specification* etc. are instances of *DataVariable* from Fig. 2. They correspond (conform) to the data types defined in this model (*Requirement*, *Specification*, *Program* etc.) shown in the bottom panel. Similarly, a *Product* model could contain product types, like *JML Specification, JavaFile*.

These two models are related together by the horizontal relationships, for example there may be a link between the data type *Specification* from the *Activity* model and the product type *JML Specification* from the *Product* model; this link conforms to the relationship represented between the concepts DataType and ProductType in Fig. 2.

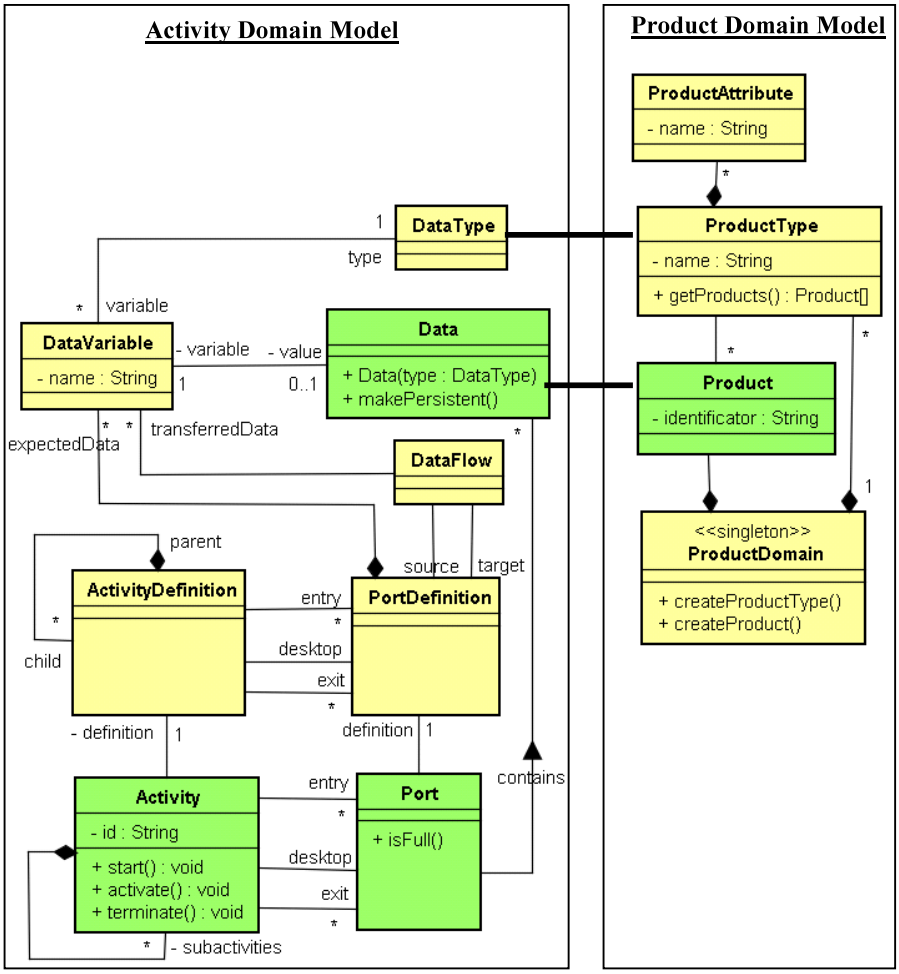


**Fig. 2.** Activity domain model vs. Product domain model

## 3 Generating code for the vertical composition

The methods defined in a concept are introduced for providing some functionality. In most cases, only a part (if any) of the functionality is defined inside the method itself, because, most often, the behavior involves the execution of some tools. The concept of *Feature* has been defined to provide the code calling the services that actually implement the expected behavior of the method. Additionally, a feature can implement a concern attached to that method, like an optional behavior, as in product line approaches.

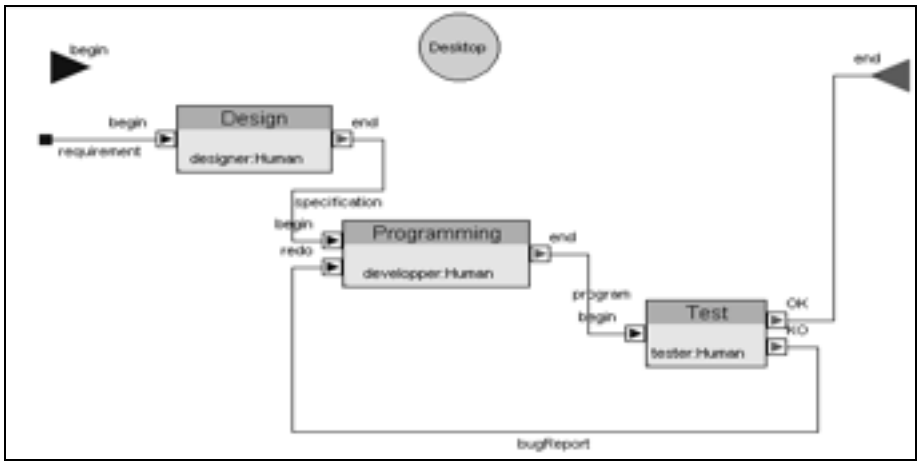


**Fig. 3.** An Activity model (fragment)

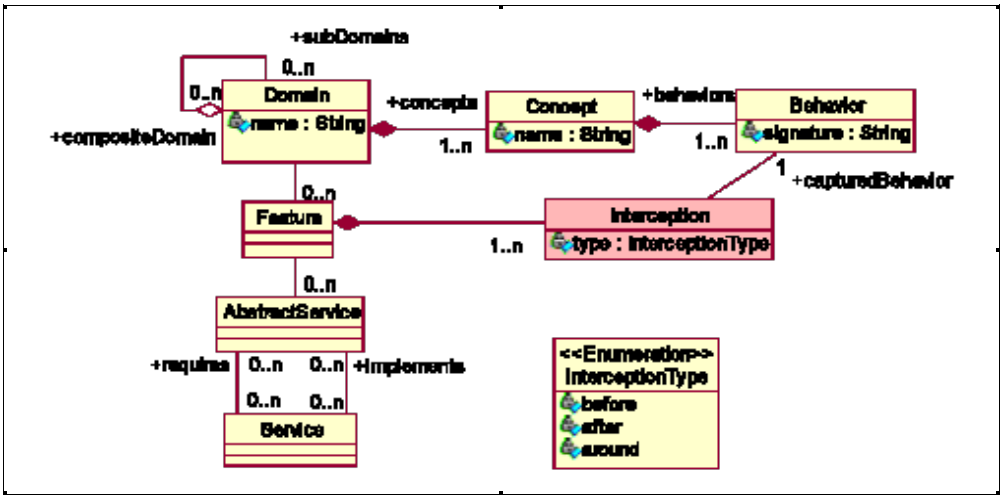


**Fig. 4.** Metamodel for the vertical composition

An abstract service is an abstraction for a set of functionalities defined in a Java interface that are ultimately executed by services / tools supporting the service (i.e. implementing its methods). For example, in Fig. 4, the method *getProducts*, in class *ProductType* is empty and it is its associated feature that will delegate the call to a database in which the actual products are stored.

More than one feature can be attached to the same method and each feature can address a different concern. The word feature is used in the product line approach to express a possible variability that may be attached to a concept. Our approach is a combination of the product line intention with the AOP implementation [12]. Moreover,

the purpose is to aid software engineers as much as possible, in the design and development of such kind of applications.

Using the Codèle tool, which "knows" this metamodel, the software engineer simply creates instances of its concepts (Behavior, Interception, Feature, Service etc.) and the tool generates the corresponding code in the Eclipse framework. As well as all Mèlusine domain models, Codèle metamodels are implemented with Java, while AspectJ, its aspect-oriented extension, is used for delegating the implementation to different tools and/or components (instances of the concept Service).

The Eclipse mappings currently used in Mélusine environment are presented in Table 1. In particular, users never see, and even ignore, that AspectJ code is generated; they simply create a feature associated with a concept behavior. A similar idea is presented in [13], where Xtend and Xpand languages are used for specifying mappings from problem to solution space and the code generation is considered to be less error-prone than the manual coding.

**Table 1.** Mapping on Eclipse Artifacts for the Vertical Composition Metamodel

| **Metamodel element** | **Eclipse artifact** | **Elements generated inside the artifacts** |
|---|---|---|
| Domain | Project | Interfaces for the domain management |
| Concept | Class | Skelton for the methods |
| Behavior | Method | Empty body by default. |
| Feature | AspectJ Project | The AspectJ aspect and a class for the behavior |
| Abstract service | Project | Java interface defining the service interface |
| Service | Project | An interface and an implementation skeleton |
| Interception | AspectJ Capture | The corresponding AspectJ code |

## 4 Generating code for the horizontal composition

In other similar approaches, as in model collaboration [14], AOP was mentioned as a possible solution for implementing the collaboration templates, among service oriented architectures (SOA), orchestration languages or coordination languages. As our approach is based on establishing relationships, it can be compared to [15], where the properties of AOP concepts are identified (e.g. behavioral and structural cross-cutting advices, static and dynamic weaving). Our intention is to identify such properties at a more abstract level, because in our approach, aspects only constitute an implementation technique. The technique we use for generating horizontal composition between domains is similar to transforming UML associations into Java code [16], but using AOP, because we are not allowed to change the domain code.

### 4.1 Meta-Metamodel for the horizontal composition

To provide an effective support for domain composition, Mélusine requires a specific formal definition and semantics. Fig. 5 shows that domain composition relies on *Horizontal Relationship*, made of connections.

A *connection* is established between a source concept, pertaining to the source domain, and a destination concept in the target domain. A connection intercepts a behavior (method) pertaining to the source concept (class), and performs some computation depending on its type: Synchronization, StaticInstantiation, DynamicInstantiation.

*Synchronization connections* are meant to synchronize the state of the destination object; they intercept all methods that change the state of the source concept, and perform the needed actions in order to change the destination concept object accordingly.

*Instantiation connections* intercept the creation of an instance of *Element* conforming to the source concept and are in charge of creating a link toward an instance of *Element* conforming to the destination concept. This instantiation connection may be performed statically or dynamically. Statically means that the pair of model elements (source, destination) are known and created before execution; dynamically means that this pair is computed during the execution, when the source object is created (eager) or when the link is needed for the first time (lazy).

To implement horizontal relationships in AspectJ, each connection is transformed into an AspectJ code that calls a method in a class generated by Codèle; users never "see" AspectJ code. In practice, the code for horizontal relationships semantics represents about 15% of the total code. The mappings towards Eclipse artifacts used by Mélusine are indicated in the table below.

**Table 2.** Mapping between Horizontal Composition concepts and Eclipse Artifacts

| **Metamodel element** | **Eclipse** | **Elements generated inside the artifacts** |
|---|---|---|
| Domain | Project | Predefined interfaces and classes. |
| Concept | Class | None |
| Behavior | Method | None |
| HorizontalRelationship | AJ Class and Java classes | - an AspectJ file containing the code for all the interceptions<br>- a Java file for each instantiation connections<br>- a Java file for each synchronization connections |
| Interception | AspectJ Capture | Lines in the AspectJ file for the interception, and a java file for the connection code. |

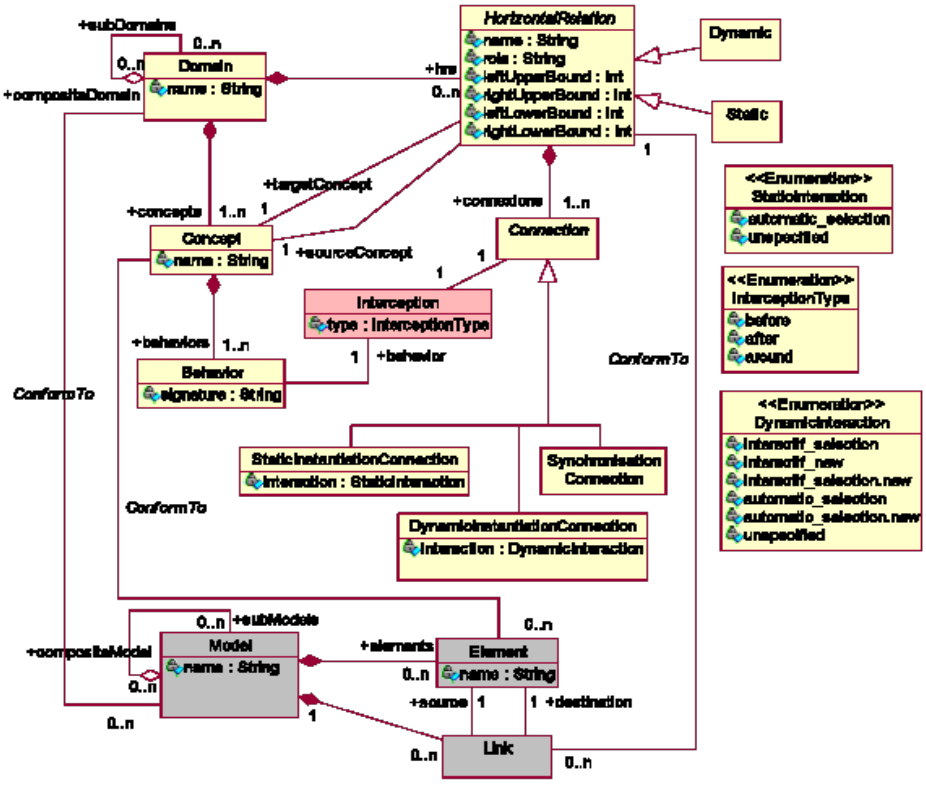


**Fig. 5.** Metamodel for the horizontal composition

### 4.2 Relationships for horizontal composition at metamodel level

Composing two domains means establishing relationships between the concepts pertaining to these domains [17]. In our example, one can establish a relationship between the concept of *DataType* in the *Activity* domain, and the concept of *ProductType* defined in the *Product* domain (see Fig. 2). The screen shot in Fig. 6 shows how this relationship is defined using Codèle tool. The horizontal relationship is defined as static, because in this specific case, the involved data types are defined in the models (*Specification* from the *Activity* model - Fig. 3 and *UMLDocument* from the *Product* model) and therefore are known before the execution. This relationship has a single connection, which intercepts the constructor of a *ProcessDataType*, with the type name as parameter, and declares that the relationship should be static and its instantiation should be done automatically, by choosing "*Static instantiation Automatic selection*" (Fig. 6).

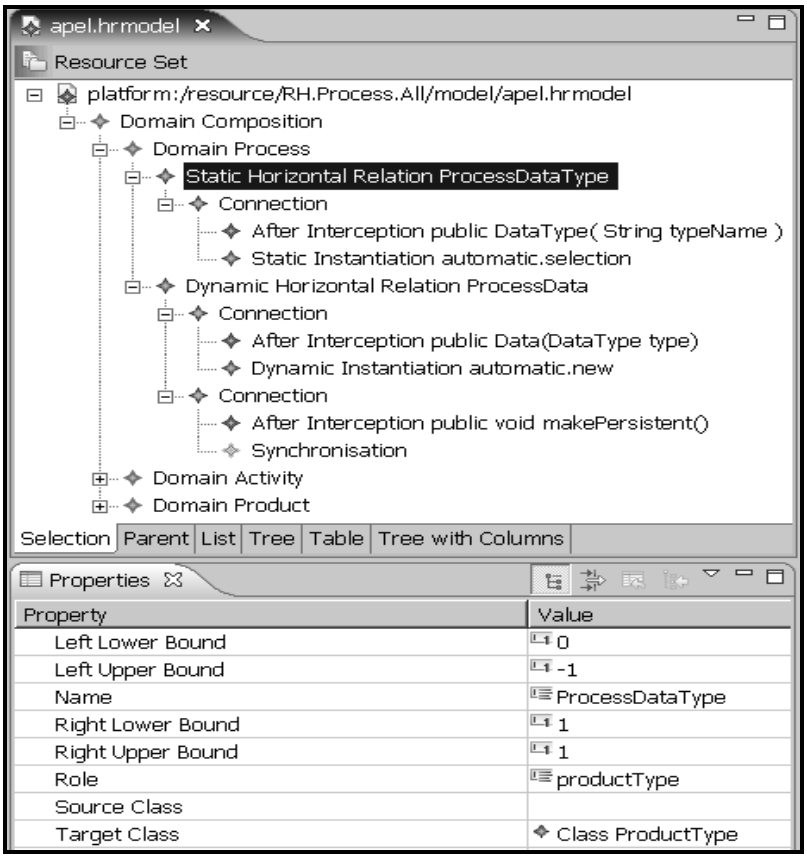


**Fig. 6.** Defining horizontal relationships at metamodel level.

It is important to mention that the system knows which concepts are visible in each domain; thus, the wizard does not allow horizontal relationships that are not valid. The interceptions are defined at the conceptual level; the developer of the composite domain does not know that AspectJ captures are generated. For the instantiation connections, the wizard proposes a set of predefined instantiation strategies, for which all the code is generated (as in our example); for the synchronization connection, the developer has to fill a method that has as parameters the context of the interception and the connection destination object.

### 4.3 Relationships for horizontal composition at model level

At metamodel level, a horizontal relationship definition is established between 2 concepts, i.e. between the Java classes that implement these concepts. However, at execution, instances of these horizontal relationships must be created between instances of these classes. At model level, Codèle proposes an editor that allows the selection of two domains (i.e. two domain models and the horizontal relationships defined as

shown in Fig. 6) and a pair of models pertaining to these domains. The top left panel lists the horizontal relationships (between the *Activity* and *Product* domains, in the example from Fig. 7). When selecting a horizontal relationship, the two top right panels show the names of the entities that are instances of the source and destination classes.

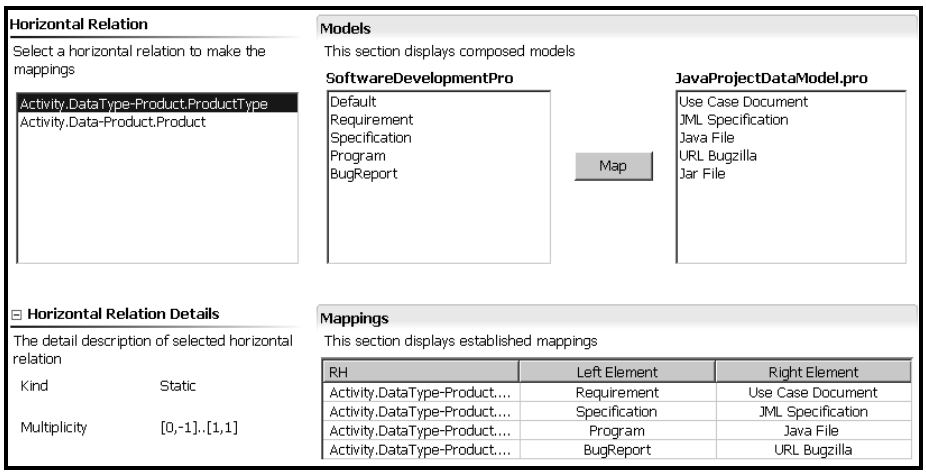


**Fig. 7.** Defining static links at model level

In our example, the *DataType-ProductType* horizontal relationship has been selected, for which one displays the corresponding instances, like *Specification* in the *Activity* domain, and *Use Case Document, or JMLSpecification* in the *Product* domain. As this horizontal relationship has been declared *Static*, the developer is asked to provide the pairs of model entities that must be linked, according to that Horizontal relationship. Otherwise, they would have been selected automatically, at run time. The bottom panel lists the pairs that have been defined. For example, the data type called *Specification* in the Activity domain is related to *JML Specification* in the *Product* domain. The system finds this information by introspecting the models and is in charge of creating these relationships at model level.

## 5 Discussion

In order to make the domain composition task as simple as possible, the metamodels presented above take into account the specificities of Mélusine domains. Consequently, the composition we realized is specific for this situation, as opposed to other approaches that try to provide mechanisms for composing heterogonous models in general contexts, generally without specifying how to implement them precisely. For this reason, many researches have tried to find out a generic approach that solves this problem, by proposing abstract composition operators, like: *match* [18], *relate* [19], *compare* [20] for discovering correspondences between models; *merge* [18], *com-*

*pose* [18], *weaving* [21] for integrating models and *sewing* [21] for relating models without changing their structure.

The elaboration of metamodels that support code generation in Codèle tool was possible after years of performing Mélusine's domain compositions. Through trials and errors, we have found recurring patterns of code when defining vertical and horizontal relationships and we have been capable of identifying some of their functional and non functional characteristics. Codèle embodies and formalizes this knowledge through simple panels, such that users "only" need to write code for the non standard functionalities. Experience shows that more than half of the code is generated in average, and that it is the generated code which is error prone, since it manages the low level technical code including AOP captures, aspect generation and so on. The user's added code fully ignores the generated one and the existence of AOP; it describes at the logical level the added functionality. Experience with Codele has shown a dramatic simplification for writing relationship, and the elimination of the most difficult bugs. In some cases, the generated code is sufficient, allowing application composition without any programming. This experience also led to the definition of a methodology for developing horizontal relationships, described in [17].

However, many other non functional characteristics could be identified and generated in the same way, and Codèle can (should) be extended to support them. We have also discovered that some, if not most, non functional characteristics cannot be defined as a domain (security, performance, transaction etc.), and therefore these non functional properties cannot be added through horizontal relationships. For these properties, we have developed another technique, called model annotation, described in [22].

## 6 Conclusion

Designing and implementing large and complex artifacts always relies on two basic principles: dividing the artifacts in parts (reducing the size and complexity of the each part) and abstracting (eliminating the irrelevant details). Our approach is an application of these general principles to the development of large software applications.

The division of applications in parts is performed by reusing large functional areas, called domains. Domains are units of reuse, primary elements for dividing the problem in parts, and atoms on which our composition techniques are applied. To support the abstraction principle, the visible part of these domains is their (domain) model; conceptually as well as technically, our composition technique only relies on domain models.

A domain is usually implemented by reusing existing parts, found on the market or inside the company, which are components or tools of various size and nature. We call vertical composition the technique which consists in relating the abstract elements found in the domain model, with the existing components found in the company. Reuse imposes that vertical relationships are implemented, without changing the domain concepts, or the existing components.

In our approach, one develops independent and autonomous domains, which become the primary element for reuse. Domain composition is performed without any

change in the composed domains, but only through so-called horizontal composition, by defining relationships between modeling elements pertaining to the composed domains.

Domains can be defined and implemented independently of each other. They are large reuse units, whose interfaces are abstract models and whose composition is only based on the knowledge of these models.

The necessity to design and implement large applications in the presence of existing components or tools led us and to develop Mélusine, a comprehensive environment, for supporting the approach presented in this paper, based on:

- Formalizing the architectural concepts related to domains, based on modeling and metamodeling;
- Formalizing domain reuse and composition through horizontal relationships;
- Formalizing component and tool reuse and composition through vertical relationships;
- Generating aspects as a hidden implementation of these composition concepts.

An important goal of our approach is to raise the level of abstraction and the granularity level at which large applications are designed, decomposed and recomposed. Moreover, these large elements are highly reusable, because the composition only needs to "see" their abstract models, not their implementation. Finally, by relating domain concepts using wizards, most compositions can be performed by domain experts, not necessarily by highly trained technical experts, as it would be the case if directly using AOP techniques.

## References


1. Le-Anh T., Estublier J., Villalobos J.: Multi-Level Composition for Software Federations. SC'2003, Warsaw, Poland, April (2003)
2. Bass L., Clements P., Kazman R.: Software Architecture in Practice. Addison-Wesley (2003)
3. Kiczales G., Lamping J., Mendhekar A., Maeda C., Lopes C., Loingtier J.-M., Irwin J.: Aspect-oriented programming. Proceedings of the European Conference on Object-Oriented Programming, 1241, 220--242 (1997)
4. Filman R.E., Elrad T., Clarke S., Aksit M.: Aspect-Oriented Software Development. Addison-Wesley Professional ISBN10: 0321219767 (2004)
5. Dave, Th.: Reflective Software Engineering - From MOPS to AOSD. Journal Of Object Technology, 1(4), September-October (2002)
6. Estublier J., Vega G., Ionita A.D.: Composing Domain-Specific Languages for Wide-scope Software Engineering Applications. Lecture Notes in Computer Science. Proceeding of MoDELS/UML Conference, Jamaica 3713, 69 – 83 (2005)
7. Monga M.: Aspect-oriented programming as model driven evolution. Proceedings of the linking aspect technology and evolution workshop (LATE) Chicago, IL (USA) (2005)
8. Ionita A. D., Estublier J., Vega G.: Variations in Model-Based Composition of Domains. Software and Service Variability Management Workshop, Helsinki, Finland, April (2007)
9. Gray J., Bapty T., Neema S., Schmidt D.C., Gokhale A., Natarajan B.: An Approach for Supporting Aspect-Oriented Domain Modeling. GPCE 2003, LNCS 2830, Springer Verlag (2003)

10. Ho W., Jezequel J.-M., Pennaneac'h F., and Plouzeau N.: A Toolkit for Weaving Aspect-Oriented UML Designs. First International Conference on Aspect-Oriented Software Development, Enschede, The Netherlands, April, 99-105 (2002).
11. Clarke S., Walker R.: Towards a Standard Design Language for AOSD. Proc. of the 1st Int. Conf. on Aspect Oriented Software Development, Enschede, Netherlands, 113-119 (2002)
12. Estublier J., Vega G.: Reuse and Variability in Large Software Applications. Proceedings of the 10th European Software Engineering Conference. Lisbon, Portugal. September (2005)
13. Voelter M., Groher I.: Product Line Implementation Using Aspect-Oriented and Model-Driven Software Development. Proc. Of the 11th International Software Prouct Line Conference (SPLC), Kyoto, Japan (2007)
14. Occello A., Casile O., Dery-Pinna A., Riveill M.: Making Domain-Specific Models Collaborate. Proc. of the 7th OOPSLA Workshop on Domain-Specific Modeling, Montréal, Canada (2007)
15. Barra Zavaleta E., Génova Fuster G., Llorens Morillo J.: An Approach to Aspect Modelling with UML 2.0. UML'2004 Workshop on Aspect-Oriented modeling, Oct. 2004, Lisbon, Portugal (2004)
16. Génova G., Ruiz del Castillo C., Lloréns J.: Mapping UML Associations into Java Code. Journal of Object Technology. 2(5): p. 135-162. Sep-Oct (2003)
17. Estublier J., Ionita A. D., Vega G.: Relationships for Domain Reuse and Composition. Journal of Research and Practice in Information Technology, 38, 4, 287-301 (2006)
18. Bernstein, P.A.: Applying model management to classical meta data problems. Proceedings of the Conference on Innovative Database Research (CIDR). Janvier 2003: Asilomar, CA, USA (2003)
19. Kurtev, I. and M. Didonet Del Fabro: A DSL for Definition of Model Composition Operators. Models and Aspects Workshop at ECOOP. July 2006, Nantes, France (2006)
20. Kolovos, D.S., R.F. Paige, and F.A.C. Polack: Model Comparison: A Foundation for Model Composition and Model Transformation Testing. GaMMa 2006, 1st International Workshop on Global Integrated Model Management. Shanghai (2006)
21. Reiter, T., et al.: Model Integration Through Mega Operations. Workshop on Model-driven Web Engineering (MDWE). Sydney (2005)
22. Stéphanie Chollet, Philippe Lalanda, André Bottaro: Transparently adding security properties to service orchestration. 3rd International IEEE Workshop on Service Oriented Architectures in Converging Networked Environments (SOCNE 08). March 2008, Ginowan, Okinawa, Japan (2008)