Softw Syst Model (2010) 9:403–424
DOI 10.1007/s10270-009-0144-1

REGULAR PAPER

Supporting fine-grained generative model-driven evolution

Theo Dirk Meijler · Jan Pettersen Nytun · Andreas Prinz · Hans Wortmann

Received: 27 August 2007 / Revised: 13 March 2009 / Accepted: 14 September 2009 / Published online: 9 January 2010
© The Author(s) 2010. This article is published with open access at Springerlink.com

Abstract In the standard generative Model-driven Architecture (MDA), adapting the models of an existing system requires re-generation and restarting of that system. This is due to a strong separation between the modeling environment and the runtime environment. Certain current approaches remove this separation, allowing a system to be changed smoothly when the model changes. These approaches are, however, based on interpretation of modeling information rather than on generation, as in MDA. This paper describes an architecture that supports fine-grained evolution combined with generative model-driven development. Fine-grained changes are applied in a generative model-driven way to a system that has itself been developed in this way. To achieve this, model changes must be propagated correctly toward impacted elements. The impact of a model change flows along three dimensions: implementation, data (instances), and modeled dependencies. These three dimensions are explicitly represented in an integrated modeling-runtime environment to enable traceability. This implies a fundamental rethinking of MDA.

Keywords Evolution · Model-driven development · Generative development · Interpretive development

Communicated by Dr. Betty Cheng.

T. D. Meijler (✉)
SAP Research, CEC Dresden, Dresden, Germany
e-mail: theo.dirk.meijler@sap.com

J. P. Nytun · A. Prinz
Faculty of Engineering and Science,
University of Agder, Agder, Norway

H. Wortmann
Faculty of Management and Organization,
University of Groningen, Groningen, The Netherlands

1 Introduction

Modern system development introduces a higher abstraction level using models. Modeling is a form of software description that is often closer to the domain expert than code. The domain expert is able to describe concepts and their relationships in a visual and intuitive form, for example, by using UML or a domain-specific modeling language. Development using models is called model-driven development (MDD); OMG has defined its model-driven architecture (MDA) [1] to support MDD. In various publications [1,2], MDD is hailed as promising to allow mapping models to different implementation platforms and to permit an easier adaptability of the models because of its higher abstraction level, thus improving model longevity when compared to code, and producing better implementation independence and a shorter time-to-market. Moreover, verification early in the development process becomes possible [3]. Various successes have been reported on the OMG website [4].

Apart from the higher level of abstraction, the evolution of systems has become an important issue. Large systems, especially enterprise systems, must be able to evolve constantly [5,6]. These systems must be adapted to changing circumstances, new products, new laws and changing organizational structures.

One important form of system evolution is that supported by reuse techniques where software components can be added or replaced. Component-based approaches such as SOA [7] use composition languages such as BPMN [8] or BPEL [9] to support large-scale software evolution.

The question is, How can reuse-based evolution be combined with the power of generative MDD [3]? The idea behind this combination goes as follows: A model of a reusable part of a system (a partial model) can be added or replaced such that:

- The impact of a replacement can be detected at the model abstraction level of other parts of the system, which may lead to other model changes
- The impacted part of the system's realization that must be re-generated can be kept as small as possible
- The impact on existing data instances, which are important assets of enterprises in such large-scale systems, can be detected
- The impact on instances that must correspondingly be upgraded can be kept as small as possible.

This paper will focus specifically on adding or replacing Model Classes, which are partial models that are at the granularity of classes (in the sense of Object-oriented systems), and on evolving a system correspondingly. Making this a possibility is the main contribution of this paper.

To support such fine-grained generative model-driven evolution, this paper will assert that three dimensions of relations must be explicitly maintained:

1. From model to realization: to trace the impact of models on realization
2. From models to data instances: to trace the impact of models on these instances
3. Between models, such that model dependencies can be traced.

The solution presented in this paper is a new hybrid approach to MDD, which combines certain essential ideas from interpretive development (such as those described by Riehle et al. [10] and Atkinson and Kuehne [11]) with generative development. In this way, it combines the advantages of interpretative development, supporting relative fine-grained partial model changes, with the advantages of generative development, supporting consistency checking and a more efficient execution.

This approach has been realized and applied in a large research project on MDD called "Nucleus," which was a research project of a large Enterprise Resource Provider (ERP) vendor that started even before the MDA was proposed [12]. Using this approach, two large beta applications have been realized in an incremental model-driven fashion in the area of product lifecycle management for airplane motors and large paper factories, respectively (see also Sect. 5). This paper can, therefore, be seen as describing some fundamental concepts of the Nucleus Modeling Framework (NMF) resulting from this research project. However, the terminology and explanation in this paper have been tailored to the current MDA literature.

This paper is structured further as follows: In Sect. 2, the meaning of generative development as in the MDA will be explained further, and contrasted with interpretive approaches. This is fundamental for the rest of the paper. Furthermore, the advantages and disadvantages of both approaches will be discussed to see why a hybrid approach may improve both concepts. In Sect. 3, two simple use-cases will be described, which correspond to the addition and replacement of a model class. These will be used throughout the paper to explain the NMF. From these use-cases, minimal requirements will be derived as guidelines for carrying out fine-grained evolution within generative model-driven development. Section 3 will also detail known limitations of the approach and define the scope of this paper. Section 4 will describe the main principles of the NMF. Section 5 will describe how the NMF fits in the larger Nucleus project, and will give a more detailed account of how the approach has been applied. Section 6 will present a discussion. As the NMF is radically different in some aspects from other modeling approaches, the soundness of its principles will be discussed. We will also discuss whether the requirements derived in Sect. 3 have been fulfilled. In Sect. 7, related work will be presented and discussed. In Sect. 8, a conclusion will be given.

2 Setting the stage: generative versus interpretive

As an introduction to the rest of this paper, the intricate relationship between generative and interpretive approaches to MDD needs to be explained. To compare generative development with interpretive development, it is necessary to understand the meaning of generative MDD; this issue will be taken up in Sect. 2.1. Generative MDD itself builds upon interpretive technologies at lower-level forms of abstraction (not model-driven); this will also be treated in Sect. 2.1. Finally, *interpretive MDD* is an alternative to generative MDD. In Sect. 2.2 an explanation will be given as to why generative MDD can potentially have important advantages as compared with interpretive MDD.

2.1 Relating generative and interpretive approaches

Since the MDA is the de-facto standard of generative MDD, we will discuss generative MDD by describing the MDA. Figure 1c illustrates one of the main principles of the MDA: An application is built by making a platform-independent model (PIM). A PIM describes the high-level logical structure of the problem domain without concerning itself with specific software platforms. When seen as a description, the PIM exists in a modeling environment and conforms to a supported language; for example, a PIM described in UML is typically object-oriented and the behavior of objects might be expressed in state charts.

The PIM is subsequently mapped to an application which is runnable on a specific software platform. Since one of the

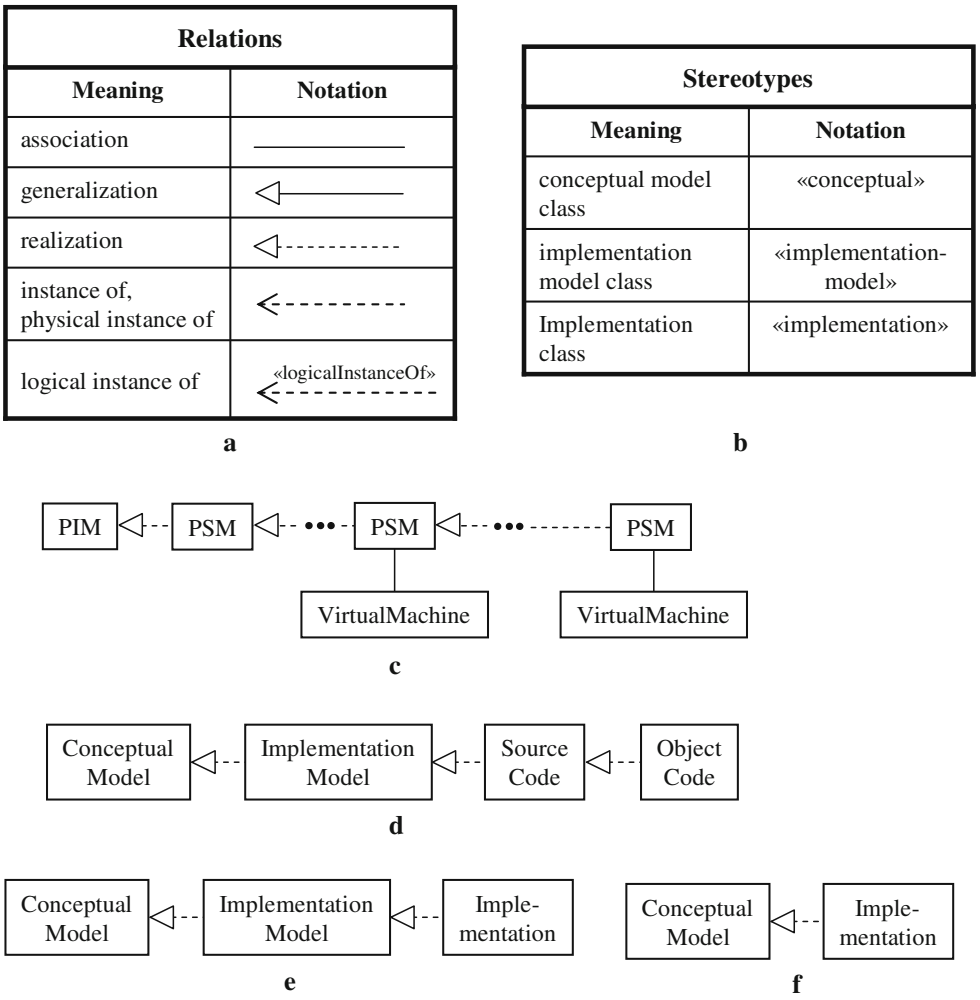


Fig. 1 Notation **a**–**b** and Mapping of PIM to PSM **c**–**f**

major purposes of the MDA is to allow for relatively simple mappings to different software platforms, this mapping is done in a series of transformation steps, from a PIM to a Platform-specific Model (PSM) and on to code. Going from a PIM to a platform is a realization/implementation of the PIM; we call this the *implementation dimension*. In Fig. 1c this dimension is modeled by a UML realization relationship (see Fig. 1a, b for notation), which also implies a dependency. The figure shows only one sequence from PIM to "final" PSM—for instance, Java byte code–but for different platforms different sequences can be followed in parallel. Thus, Fig. 1c only shows parts of a possibly more complex picture.

Figure 1d shows our approach: The PIM is called the Conceptual Model; it is mapped to a more platform-specific model called the Implementation Model. Java is our target platform and the Implementation Model is mapped to Java source code, which is compiled to Java byte code. In the coming discussions it will not be necessary to consider all the elements of Fig. 1d; Fig. 1e shows a simplified view where source and object code together are seen as the implementation (i.e., the developer does not edit Java byte code). Because realization is transitive, we have in some cases simplified further and only used the elements shown in Fig. 1f. The stereotypes given in Fig. 1b match the elements of Fig. 1e and will be used in later figures.

The mappings from PIM to PSM and from one PSM to another PSM are steps that generate models and software. Consequently, the MDA approach is essentially generative, but generated code other than machine code must still be interpreted by some virtual machine; thus interpretive technologies are also applied here.

In a situation as described in Fig. 1c, the internals of the interpreter are typically not the concern of the developer and hence the developer sees the interpreter as being able to understand the PSM (e.g., Java byte code). The interpreter, in our case a JVM, in a sense defines where the job of the developer "stops" and the runtime environment "takes over"; the JVM appears as a black box.

The level of abstraction provided by a virtual machine to provide further generation steps is fundamentally an arbitrary choice. One might imagine a VM that interprets Java source

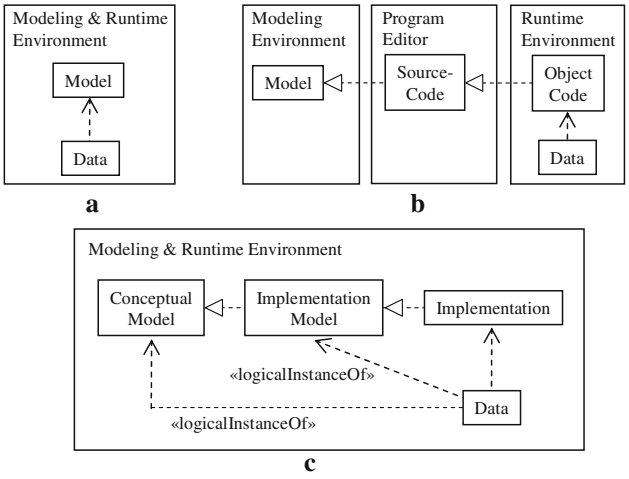


Fig. 2 Interpretative (**a**) Generative, (**b**) and (**c**) a hybrid approach

code "directly," and one might argue that such an interpreter is more interpretive than the standard JVM, since it operates at a higher level of abstraction.

When referring to an interpretive approach in this paper, we will limit ourselves to an approach where the PIM is interpreted and there is no additional execution language; in effect, the modeling language is the execution language. Interpretive approaches are described by Riehle et. al. [10], Atkinson and Kühne [11], Mellor and Balcer [13] and others.

In contrast, a *generative* approach involves at least one generative step. Figure 2b gives a typical example: The model (e.g., a PIM) is mapped to source code which can be adjusted by the developer; source code is compiled to object code and the developer must start or restart the application to experience the changes.

Figure 2a shows the situation for an interpretive approach where the modeling and runtime environment are the same. An object-oriented modeling language typically offers a way to model classes and instances of the classes; this is demonstrated in Fig. 2a (the instances of the classes constitute the data). Since the modeling and runtime environment are the same, both classes and class instances are explicitly or implicitly manipulated at runtime; in this respect, it is related to meta-programming [14].

Figure 2b shows the generative approach where the developer relates two languages: the modeling language (e.g., UML) and the programming language. The developer does not edit the target code of the compilation, and debuggers communicate with the developer on the level of the programming language. In the subsequent section, these two approaches will be compared.

The approach presented in this paper is a hybrid approach in which certain essential ideas from interpretive approaches have been combined with generative development. Figure 2c shows that in this approach modeling and runtime environment are also integrated, where modeled classes are explicitly represented, and actual data objects can be directly related to their modeled classes through the `«logical-InstanceOf»` relation.

2.2 Comparing generative and interpretive approaches

When comparing generative and interpretive approaches, the following observations can be made:

- Since an interpreter executes statements line by line, interpreters principally make the replacement of any set of lines possible, and thus, they allow a system to be adapted as long as none of these lines is currently being executed. In other words, interpreters fundamentally support evolution in the sense that even a small model change is immediately visible.
- The advantage of generative approaches is that they can lead to faster execution [15,16] and better error prevention (e.g., if statically typed [17]). Moreover, in the generation step, an optimal target language can be selected, for instance, a language for process execution (such as BPEL), or a language for query execution such as SQL.
- Related to that, interpreted approaches tend to use their own data storage, for example, tables consisting of tuples for each property value, including minimally the property type and property value [18]. This is more inefficient with respect to storage space, since for generated types all single-valued property values can be stored in one tuple. Moreover, tables can only be queried from within the interpreter environment.
- A modeling language interpreter may itself be implemented in some programming language (e.g., Java, Lisp) that offers an execution platform. By contrast, in a generative approach, the set of used target languages together forms the execution platform. The execution platform of an interpreter can be adapted by porting the interpreter to a different execution platform, for instance, porting the interpreter from a Java implementation to a C# implementation. The execution platform of a generative approach can be adapted by selecting a different generator. Moreover, new platforms can be supported by adding new generators; this is often less laborious than porting an interpreter to a new platform. Thus, with respect to portability to different execution platforms, the generative approach is potentially simpler and less error-prone than the interpretive approach.

Since generative approaches distinguish between the programming language and the execution language, a distinction in granularity will appear between the changed statements in the programming language and the generatively changed statements in the execution language. In modeling

environments, this can even lead to "big-bang" development approaches where replacement of some model parts can lead to the full replacement of a runtime system or a component.

In interpreted approaches, changes can be applied ad hoc, easily leading to errors due to dependencies. For example, when removing a certain method, runtime errors can occur when a dependent class still invokes such a method. A generation or compilation step can be used to detect such errors.

Given this comparison, a hybrid approach is warranted that combines the advantages of the execution efficiency, portability support, and inconsistency checking of generative approaches with the possibility of fine-grained partial model replacement (and thus better support for evolution) of interpretive approaches.

The hybrid approach presented in this paper is similar to an interpreted approach in the sense that it supports fine-grained replacements. It is generative in the sense that the replacement corresponds to class models and their generated code. Figure 2c shows that the presented hybrid approach integrates the modeling environment and runtime environment as in interpretive approaches. From this figure it can be seen that the improved support for fine-grained evolution will come at a price: the approach includes a preferred runtime (in our case Java-based) platform in which the integrated environment is realized. The possibility to generate different software platforms from one model, as explained for the MDA, is in principle still possible; however, by generating a platform other than the preferred one, fine-grained evolution will no longer be supported.

3 Use-cases, requirements and limitations

In this section, requirements for fine-grained generative model-driven evolution will be derived using two simple use-cases which correspond to the addition and replacement of a class model. One such use-case comprises the addition of a model class; another use-case comprises the replacement of a model class by another model class. The idea is that these are the most essential "change operations" that are urgently needed. Many larger system-restructuring operations are based on these essential operations. Larger system-restructuring operations that include class removals or other fundamental changes, such as class merges, require more advanced mechanisms, since there is no straightforward (automatable) way to associate existing instances with a fundamentally new set of classes. Such advanced mechanisms are the subject of further study.

3.1 Use-cases

Suppose a company has purchased a brand-new system for handling their customer relations. This system features a model-driven approach that allows for continuous extension by adding new classes. In the initial system, relatively general concepts were modeled and implemented, such as `Customer`. The initial system was implemented in Java.

3.1.1 Terminology

Since the subject of this paper is evolutionary MDD, *partial* models of the system and the corresponding reality, such as the model of the class `Customer` plus its corresponding implementation, are of interest. Since we will be distinguishing explicitly between a partial *model* of a single class and the *implementation* of that class in the runtime system in what follows, we will use the terms "model class" and "implementation class" for these, respectively. A distinction will also be made between a model class that is closer to a model of the concept (such as `Customer`) and a model class that is closer to a model of the implementation. This distinction is similar to the PIM/PSM distinction for single modeled classes. The first modeled class will be called the Conceptual Model Class (CMC), the second the Implementation Model Class (IMC) (see stereotypes in Fig. 1b).

If one considers a type to be "a specification of the general structure and behavior of a domain of objects without providing a physical implementation," [19], then what we call a CMC starts out as a type (which cannot be instantiated) and is made a class by adding an implementation that allows instantiation.

Using these terms, the initial system encompasses the model class and implementation class `Customer`. The example encompasses two use-cases.

3.1.2 Use-case 1: extension

The company has a need to extend the system to include the extra class `Premium Customer`. A premium customer has specific rights, such as buying goods on credit and obtaining special discounts. Moreover, special information is maintained about a premium customer, such as a description of his/her main interests.

Enabling evolution means that the company is enabled to model this new `Premium Customer` class and to develop the corresponding implementation without touching the rest of the system. The company leads the model class through a development process; this includes adding further details about the corresponding class, such as possibly adding hand-written Java code. Thus, an implementation class is created for this `Premium Customer` model class, which can be separately compiled and added to the existing system.

Figure 3 describes a generative extension in the form of a picture story. Picture (1) of the figure presents the `Customer` model class, the corresponding implementation class, and the possible instances; for example, data about

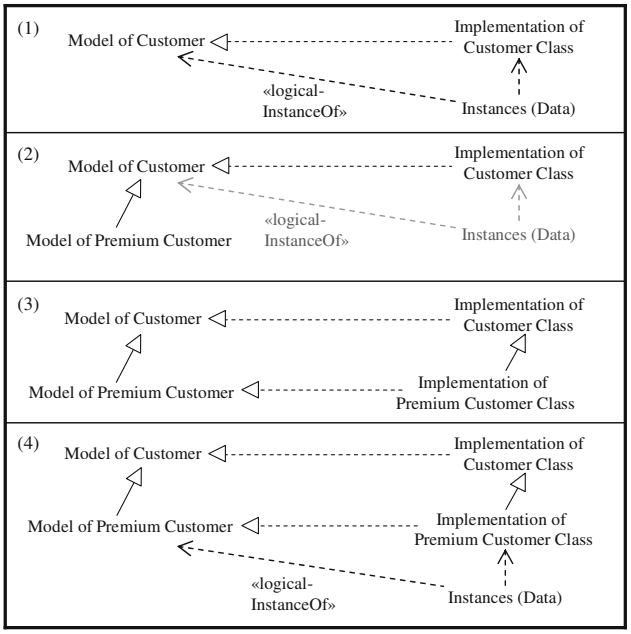


Fig. 3 Picture story of generative extension

specific customers in the existing system in model-driven terms. The relationships displayed imply that the implementation and the instance data are considered to be dependent on the model class.

Picture (2) of the figure presents a *modeled* extension of the system. A new class for `Premium Customer` is modeled. The `Premium Customer` model class models a subclass relationship with the `Customer` model class. The `Premium Customer` model class may introduce new properties and behavior such as the property `main interest`. The fact that the existing instances of the class `Customer` are grayed out indicates that these instances are not impacted by the extension with a new model class.

Picture (3) of the figure presents the generation of the implementation class `Premium Customer`. This generated implementation class has to be related to the other implementation artifacts in the existing system. In particular, it needs to become a subclass of the implementation class `Customer`.

Picture (4) of the figure represents the situation where, after the extension with the new class `Premium Customer`, new instances for this class may be instantiated.

The given example conforms to the open/closed principle as first introduced by Bertrand Meyer [20]: software entities (e.g., classes, modules, and functions) should be open for extension but closed for modification. In our case, the extension is done by modeling a new subclass and only the newly modeled subclass needs to be considered (e.g., tested).

Use-case 1 is called an *extension*. In model-driven evolution, not every *addition* of modeled information with a corresponding adaptation of the implementation is regarded as an extension. It is only an extension when the rest of the implemented system is not impacted. Here no other model classes or implementation classes are impacted, nor are any existing instances impacted. The addition of new properties or behaviors to an existing class is, therefore, not considered as an extension but as an adaptation. This will be treated in the subsequent use-case.

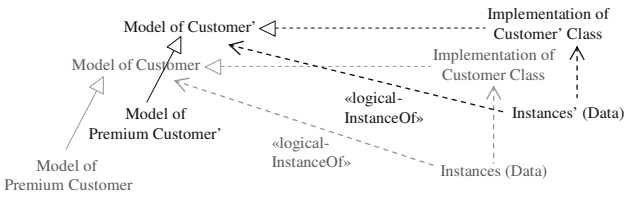


Fig. 4 A simple use-case of generative adaptation

3.1.3 Use-case 2: adaptation and replacement

Figure 4 characterizes a use-case of model-driven generative *adaptation* and corresponding replacement. The use-case starts at the final state of the use-case just described. In this second use-case, the property `main interest` has been added to the `Customer` model class. As a result, the corresponding implementation class and the existing customer instances must be updated. To prevent a double implementation the developer may also want to remove this property in the `Premium Customer` model class and implementation class.[1] Note that the impact is recursive: the change in the model of the dependent class may again have an impact on its implementation class, instances, and dependent classes (this is not explicitly shown in the figure).

Figure 4, moreover, shows how adaptation leads to replacement. The new `Customer` model class (`Customer'`) with the new modeled property `main interest` replaces the old `Customer` model class. The implementation class `Customer' Class` will replace the implementation class `Customer Class`. Similarly, the new instances (Instances') of the `Customer' Class` – with the main interest property now filled in—will replace the old instances. Conceptually, the values of other properties in the new instances will be the same as in the old instances although the implementation could be changed. Also, the new model class of `Premium Customer` i.e., `Premium Customer'` will replace the old model class. This will again lead to replacement of the corresponding implementation class, data, and dependent model classes (when available). Thus, the replacement process is recursive along the lines of dependency.

[1] As will be detailed in Sect. 3.3, no special support is offered to the developer so that he/she will know what the impact of the change of the `Customer` Model class will be on the `Premium Customer` Model class.

3.2 Deriving requirements

In this section a set of requirements will be derived for enabling fine-grained evolution, as inspired by the use-cases of Sect. 3.1. These use-cases introduce major starting points that will be used throughout this paper:

- Extending upon or improving upon a model of a certain enterprise system can be seen as introducing a new model class or *replacing* a model class by a new partial model. Removing a model class will not be covered.
- Any system change is seen as a change in a set of model classes and their corresponding implementation classes. Even independently defined behavior, such as an activity, is identified through a model class and a class [21]. Thus, a change is modeled as a replacement of certain model classes. Changes such as data storage and the user interface are regarded as secondary, since they will be based on the changes in the model classes and implementation classes. The same can be true for component interfaces [22] where each class is mapped to a software component.

The principle of viewing changes in terms of changes to classes corresponds to describing and implementing systems in terms of objects [23]. The principle of viewing adaptation as replacement is consistent with the way change is handled in version management systems [24]. Replacement can, therefore, be viewed as going from an older version to a newer version. Such replacement will lead to an impact that must be managed and that may also lead to newer versions of other entities (model classes, implementation classes and instances).

As will be seen in the following, the requirements that enable adaptation through replacement encompass the requirements enabling extension. To derive these requirements, Use-case 2 will first be described in abstract terms, ignoring the precise names of the classes and their relationships.

Figure 5 presents Use-case 2 in abstract terms. It can be described as follows:

- Let `Mc` denote a model class that is replaced by model class `Mc'` (e.g., adding a property).
- The corresponding implementation class is called `Ic`; after the replacement of `Mc`, `Ic` will be replaced by `Ic'`.
- `Mc` itself models the reuse or specialization of another model class `Ms`. The implementation class `Is` is the implementation of Ms. `Ic'` must again inherit from `Is`.
- Corresponding to `Mc` and `Ic` are the data objects `Dc`. Each data object `Dc` must be replaced by `Dc'`.
- The model class `Md` models a dependency with `Mc`; thus, `Md` has to be replaced by `Md'`. This replacement will be done for all dependent partial models. This is a recursive process, proceeding with the replacement of `Md` by `Md'`, Id by `Id'` and `Dd` with `Dd'`, and so on.

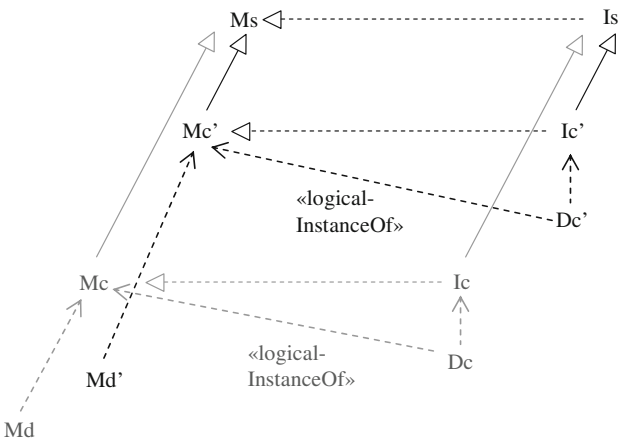


Fig. 5 Abstract characterization of generative adaptation

In order to support this abstract use-case of the adaptation of `Mc` to `Mc'`, the following requirements must be fulfilled:

(a) Being able to replace the implementation class `Ic` by the implementation class `Ic'` and hook up `Ic'` with the implementation of the rest of the system. This means that `Ic'` must faithfully implement its (inheritance) relationship to `Is` (as indicated in Fig. 5 by an arrow from `Ic'` to `Is`).
(b) Enable the runtime system to start using and, especially, to instantiate the new/ replaced class.
(c) Being able to find the instances (data objects) `Dc` and replace them by `Dc'`.
(d) Being able to find the dependent model classes `Md` and replace them by `Md'`.

In a case where `Mc'` does not replace `Mc` but `Mc'` is newly introduced, we speak of extension. In that case, only Requirements (a) and (b) above are relevant. Thus, the requirements enabling adaptation through replacement do encompass the requirements enabling extension.

The Requirements (a), (b), (c), and (d) can be summarized as follows. To enable the generative replacement of model classes, impact management of such replacement must be supported along three dimensions:

- Along the implementation dimension: as covered by Requirements (a) and (b).
- Along the instantiation dimension: as covered by Requirement (c).
- Along the modeled dependency dimension: as covered by Requirement (d).

3.3 Scoping issues and limitations of this paper

As mentioned in the introduction, this paper focuses on the (technical) replacement of model classes in an existing model of a system. One limitation of our approach is that it treats only those forms of evolution where the class structure is not broken; thus, class removal is not supported. This is especially due to Requirement (c) in Sect. 3.2, since re-assigning existing instances to classes in case of removal is not straightforward, and therefore no solution is offered for this situation.

Many aspects that are also of relevance to model-driven software evolution, and software evolution in general in a wider sense, are beyond the scope of this paper, but could be used in combination with our approach. These are aspects such as

- How to go from changed use-cases to adapted models (see e.g., [25]).
- How to re-engineer a system [26] and even how to precisely determine the impact of a change, given that an infrastructure as described is available.

This paper (as is the case for NMF) focuses on functional and non-crosscutting changes. These are changes for non-crosscutting concerns, which can be contrasted with crosscutting concerns as defined in the literature on aspect orientation [27]. Crosscutting concerns can only be realized across the normal modular implementation of software systems. An example of what is *not* covered in this paper is the introduction of an improved form of caching, such that database retrieval can be minimized.

Furthermore, this paper focuses on enabling the identification of impacted elements and their replacement and not on determining the precise impact of the change on these elements. As shown in Use-case 2, what is covered is the *potential* impact of the addition of a property to the model class `Customer` on the model class `Premium Customer`. What is not covered for this use-case is the explicit support indicating that the addition of the property `main interest` may lead to a removal of the same property on the `Premium Customer` model class. This is covered by other approaches [28,29]. Again, such approaches could be combined with ours.

A scope limitation of this paper is related to meta-modeling. Meta-models are used in order to formally describe the model structures and relationships allowed in a modeling environment. Meta-modeling plays an important role in the NMF, because it is used to describe how relationships—and thus tracing—between models, and between models and the rest of the system, can be realized. However, due to size limitations, meta-modeling has not been covered in full in this paper.

Large-scale enterprise systems are often built as decoupled multi-tier architectures [30], for example, consisting of a UI tier, a business logic tier and a data persistence tier. Each of the tiers uses different technologies. For instance, a front tier may use an XML-based browser technology. A business logic tier may use process modeling and execution, and business objects implemented using programming languages. In the data persistency tier data are stored and accessed using database technology. In such a multi-tier architecture a model can be used to keep the different layers consistent [30]. This same principle has been used in Nucleus. Again, this part of the NMF cannot be discussed here due to space limitations.

Given such a three-tiered approach, three different implementation aspects will be generated from a Conceptual Model: the logic aspect, the user interface aspect, and the data persistence aspect. This paper will only focus on the relationship between the conceptual model and one specific implementation aspect, namely the business logic. This paper does not cover mapping to component approaches, such as service-oriented architecture.

Finally, the scope of modeling in the NMF software is worth mentioning here. In the NMF, the conceptual model plays a central role and it generates to Java. However, the system is not completely described by models. The main elements of the system, such as its classes, attributes, and relationships, are modeled. For some behavioral aspects, namely workflow activities, so-called "visitors"[2] and code generators themselves, Domain-Specific Modeling Languages (DSMLs) are successfully applied [31], but, for other forms of behavior, Java methods are implemented by hand. The underlying reasoning is that creating a DSML only pays off when there are sufficient occurrences where it can be applied and if and when these occurrences lend themselves to expression in some language due to their similarities. Using modeling instead of Java for all other behavior descriptions requires behavioral models at a level of detail that makes the models just as complex as the code and, therefore, does not contribute to programming productivity.

4 Principles of the nucleus modeling framework

This section will summarize the basic principles of the Nucleus Modeling Framework (NMF). As based on the requirements described in Sect. 3.2, the NMF is based on adding or replacing model classes and, from there, supports traceability across three dimensions: Implementation, Instantiation and Model Dependency. This starting point is characterized in Fig. 6 by placing a main Conceptual Model Class (CMC) as the corner of the three dimensions of the NMF;

[2] Models of visitor [38] behavior are somewhat similar to behavior specifications used in attribute grammars.

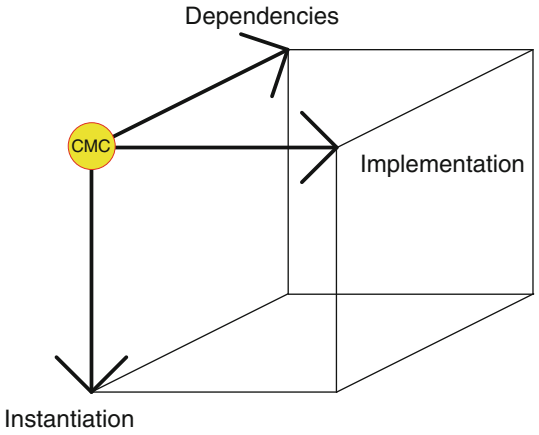


Fig. 6 The conceptual model class (CMC) and the three dimensions of traceability

subsequent subsections will extend this picture. In the following subsections, the support for identification and replacement along these dimensions will be detailed further.

4.1 Representing the model class as an object

Tracing replacements of partial models (model classes) with the corresponding partial replacements of the implementation and of the instances in the runtime system (i.e., traceability along the Implementation and Instantiation dimensions) implies that replacements of the partial models are, so to speak, shadowed in the runtime system. Riehle et. al. [10] call this kind of shadowing "causal connectedness" between the modeling environment and the runtime environment.[3] This must clearly lead to some form of integration between the modeling environment and the runtime environment [10,11,32]. The NMF carries out a full integration, in the sense that all *model classes are represented as runtime objects*. This full integration is especially useful in supporting traceability along the instantiation dimension between model class and instance, in order to fulfill requirement c) in Sect. 3.2. Thus, the CMC in Fig. 6 is really represented as an "object" corresponding to standard object-oriented principles [33].

We will introduce some basic terminology before we discuss in further detail the representation of model classes as objects. An "*object*" is—corresponding to standard object-oriented principles [33]—an entity in a computer information system with behavior that can be invoked through messages, leading to method invocations. Methods can be invoked to access an object's state information and adapt it. Moreover, special-purpose methods can be invoked; for example, for a `customer` object, a method can be invoked to ask for the customer's closest branch office or shop.

An object can have one or more descriptive entities called classes. The object will have an instantiation relationship with the class from which it was instantiated. This relationship is often called instanceOf (the instanceOf relationship comes in several flavors [34]).

Representing a model class as a runtime object means several things:

- The model class object represents the modeling information, for instance, details of the `PremiumCustomer` class.
- The model class object can be accessed and modified in order to develop a complete model. The model class object may understand method invocations such as "check model" and "generate code."

A model class describes different aspects of its instances such as the properties and relationships, but also its behavior implementations. That a model class can be represented by an object is best shown with two UML representations of the model class in the NMF. In Fig. 7 the CMC for `Customer` is represented as a class diagram; in Fig. 8 this same information is represented as an instance (or object) diagram. The model class contains various stereotypes, for example, `«conceptual»` and `«property»` and tagged values such as `{implementationModel = CustomerImpl}`. The precise meaning of these stereotypes is not of relevance at this point.

Setting aside the precise semantics of the aforementioned stereotypes, a comparison of Figs. 7 and 8 shows how stereotypes are being used [35]. A stereotype in the class diagram in Fig. 7 corresponds to the type of the corresponding object in Fig. 8. For example, the stereotypes `«conceptual»` and `«property»` correspond to the implementation classes `CConceptual` and `CProperty` of the corresponding objects in Fig. 8.[4] Not all details of Fig. 7 are represented in Fig. 8, to prevent clutter; the tagged values and the parameters of the `buyProduct` operations, for instance, are omitted.

4.2 Representing the implementation dimension

A partial model replacement must lead to a corresponding replacement of the implementation. This is the first part of Requirement a), found in Sect. 3.2. This is supported by the implementation dimension as will be discussed in this subsection. As mentioned in Sect. 3.3, this section only treats the

[3] Causal connectedness has been defined by D. Riehle et. al. [10] as follows: "A modeling level is causally connected with the next higher modeling level, if the lower level conforms to the higher level and if changes in the higher lead to according changes in the lower level." The runtime system is seen as Level 0, the modeling system as Level 1.

[4] These classes are implementations for certain metamodel classes: `Conceptual Model Class` and `Property`.

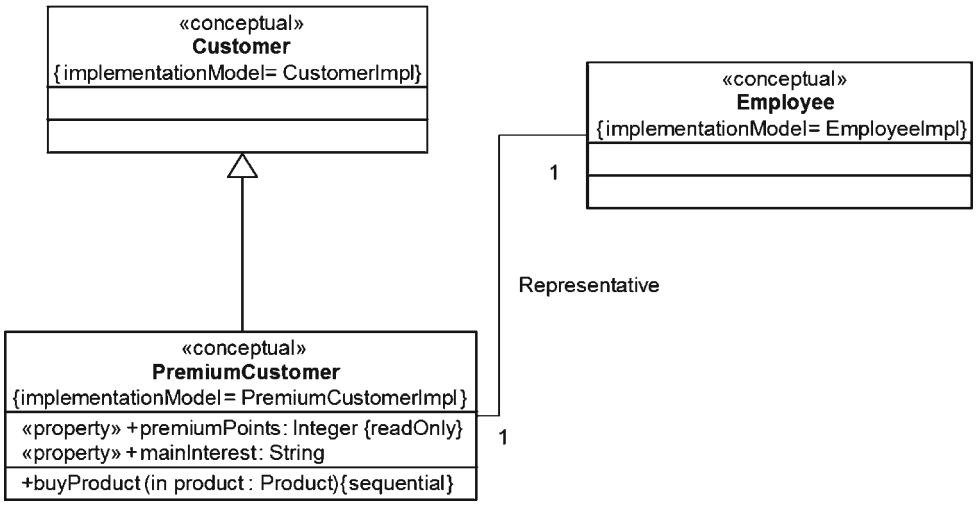


Fig. 7 Model presentation of a CMC

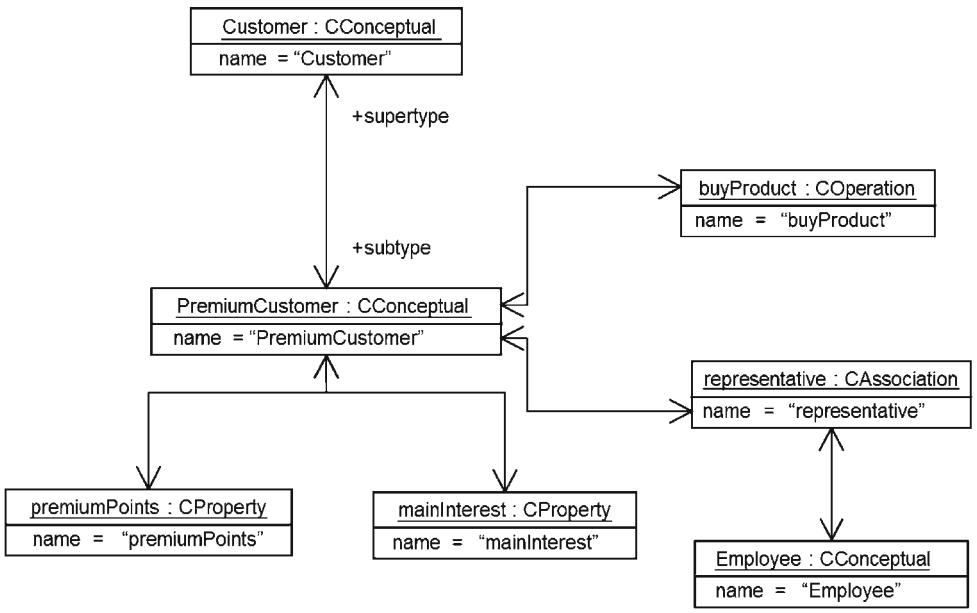


Fig. 8 Object representation of a CMC

relationship between a CMC and the "logic" implementation aspect. Other implementation aspects that are also derived from a CMC are the user interface and the data persistence aspects.

The implementation dimension for each implementation aspect, specifically the logic aspect as described in this paper, is represented by three kinds of elements (see Fig. 9). The first element is the CMC (e.g., `Premium Customer` CMC) which is an implementation-independent model, similar to a "Platform-independent Model" (PIM) in the MDA [1]. This CMC is the basis for the other implementation aspects as well and includes those aspects of a class that are common to those different implementation aspects. The second element is the Implementation Model Class IMC (which is similar to a "Platform-specific Model" [PSM] in the MDA), which

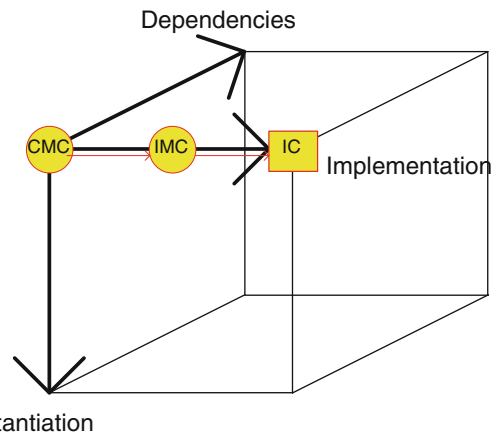


Fig. 9 The implementation dimension

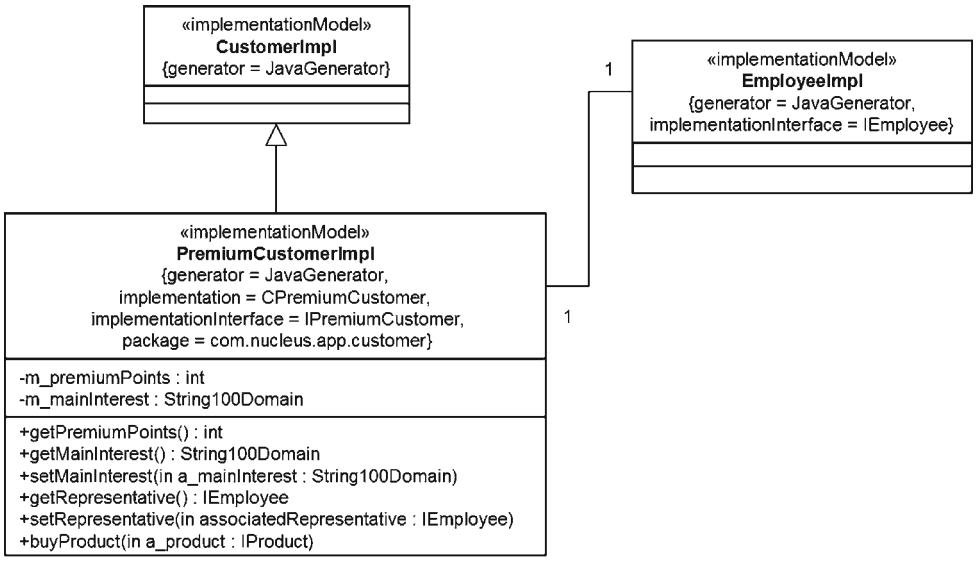


Fig. 10 IMC of `PremiumCustomer` for Java

models specific information about the corresponding implementation class. In principle, this could be done in multiple stages of refinement, corresponding to different degrees of platform dependence or independence (see also [36]), but for simplicity's sake only one refinement stage has been used in Nucleus. For mapping to a Java platform, such an IMC describes fields and methods of Java classes. The last element of the implementation dimension is the implementation class itself, which, in the case of a Java platform, consists of a Java class and a Java interface. For classes that are under development, both the Java code will be available and, after compilation, the byte code as well. In certain cases, namely when reusing classes developed by third parties, code may not be available but only binaries (Java byte code).

As represented by the thin line in Fig. 9, the implementation traceability in the NMF is realized through forward references: from CMC to IMC and from IMC to IC. Using these forward references, a replacement of the CMC will lead to a corresponding replacement of the IMC and, subsequently, of the implementation class. This will be discussed in further detail using the use-case example.

Figure 7 presents the CMC of the class `Premium Customer` as indicated by the stereotype «`conceptual`». Properties of premium customers are `premiumPoints` and `mainInterest` as indicated by the stereotype «`property`» (similar to the property concept for EJB applications [37]). Properties can either allow the reading of a property value only, or both the reading and writing of a property value. The UML readOnly property-string is used to indicate this (see Fig. 7); if omitted, then both read and write are allowed. The value of the `premiumPoints` property can only be read. The CMC does not indicate how such properties will be implemented. The CMC refers to the IMC using the tagged value `{implementationModel = PremiumCustomerImpl}`.

Figure 10 presents an IMC `PremiumCustomerImpl` for the implementation class `Premium Customer`. This model is again represented as a runtime object, describing the implementation of the class `Premium Customer` as a Java-class implementation. For each property of the CMC `Premium Customer` (e.g., the property `premiumPoints` as presented in Fig. 7) a corresponding Java field is defined and, where appropriate, also a setter and getter will be defined. In our case the getter `getPremiumPoints` is defined since the property `premiumPoints` has been defined as readable in the CMC. The IMC can, in large part, be generated from the CMC.

The IMC is used in two ways in the NMF:

- It offers mapping information on how CMCs must be mapped to an implementation. This is needed since there is no default mapping from names of CMCs to names and locations of implementation classes. For example, the tags `implementation, implementation Interface` and package are used to refer to the implementation class name, interface name and package. This information must be added by hand by a developer.
- It offers information about which generator to use when an implementation class is to be generated. The IMC for `Premium Customer` as presented in Fig. 10 has the tag `{generator = JavaGenerator}`, which indicates that the Java-class generation will be applied.

Figure 11 presents one page of a Java class that may be generated from the `PremiumCustomerImpl` model class as presented in Fig. 10. The Java-class file may be further filled in by a developer to implement specific logic; in this page no specific code has been added.

An important function of this forward reference is that the *replacement* of a CMC leads also to the replacement of the

```
/*
file : @(#)CXAppPremiumCustomer.java
*/
package com.nucleus.app.customer;
import com.nucleus.app.product.IXAppProduct;
import com.nucleus.efc.enterprise.IXEafEmployee
import
com.nucleus.efc.kernel.common.IXMfInstanceType;
import com.nucleus.sys.dom.string.String100Domain;
/**
 * PremiumCustomer. A customer who is set apart,
since (s)he is a
regular
 * buyer, and shows special interest in the
products of the company
 * @author Nucleus APP, Theo Dirk Meijler
 * @version 1.0
 */
public class CXAppPremiumCustomer extends
CXAppCustomer implements
IXAppPremiumCustomer
{
/**
 * The amount of premium points earned by this
customer.
 * @generated
 */
private int m_premiumPoints = 0;
/**
 * The main interest of this customer.
 * @generated
 */
private String100Domain m_mainInterest = null;
/**
 * field for the Representative association
 * @generated
 */
private IXEafEmployee m_representative;
/**
 * Initializer of type 'PremiumCustomer Type'.
 * @param a_type The type of this object.
 * @generated
 */
public void
initXAppPremiumCustomer(IXMfInstanceType a_type)
{
      initBMfInstance1(a_type);
}
```

Fig. 11 One Page of the `CPremiumCustomer` Java Class

IMC, and to a replacement of the corresponding implementation class. This is the first step towards fulfilling Requirement a) in Sect. 3.2: a partial model replacement can lead to a corresponding replacement of a specific part of the implementation. It should be noted that forward references are possible due to the concept of a preferred implementation platform as already mentioned in Sect. 2.2: the referred IMC corresponds to this preferred platform (i.e., it is that implementation platform in which the modeling environment itself is implemented).

4.3 Representing the instantiation dimension

In the instantiation dimension, two main requirements, as discussed in Sect. 3.2, are relevant: How to instantiate (create new data objects) for newly modeled classes (Requirement (b)); and how to keep track of the relationship between a data object and the model class that created it, such that impacted data objects (i.e., the instances) can be identified when the model class is changed (Requirement (c)). The two principles that support these requirements in the NMF will be discussed in the subsections that follow.

4.3.1 Instantiation: the CMC object as a factory

According to Sect. 4.2, a CMC refers to a single IMC, which again refers to an Implementation Class. Thus, indirectly, once the complete development process for such a model has been completed, for a CMC (of a class) the corresponding implementation class is known. This is used to enable the use of the CMC as a factory object. The CMC object can be requested to create instances. Note that the factory pattern is used [38]. A CMC object will create an instance by finding out through the corresponding IMC which class must be instantiated. Thus, by instantiating the CMC object `Premium Customer`, an object will be an instance of the implementation class `CPremiumCustomer` due to the fact that the IMC of `Premium Customer` refers to the implementation class `CPremiumCustomer`.

Through this factory mechanism, a CMC can be added or replaced, and subsequently used and instantiated in the runtime system. In other words, Requirement b) of Sect. 3.2 can be fulfilled. The set of CMC objects for which an implementation has been generated represents the catalogue of classes that can be instantiated.

4.3.2 Logical vs. physical instantiation

When instantiating the CMC a data object is created as described in the previous subsection. In principle, this object is now both an instance of the CMC and of the Implementation Class. To enable the traceability between CMC and its instances as demanded by Requirement c) of Sect. 3.2, the instantiation relationship between the data object and the CMC object is explicitly represented as an object-to-object relationship.

The result is a double instantiation as is symbolically presented in Fig. 12, by means of the thin arrows from DO to CMC and from DO to IC. Figure 13 presents the double instantiation for the example. In agreement with [11,32], these two relationships are called logical instantiation and physical instantiation (in the other figures, this relationship is shown without any stereotype). The relationship between the instance and its CMC is a logical instantiation relationship, since the CMC logically describes the instance. The generated implementation class is “physically” used to carry out the instantiation and, consequently, there is a physical instantiation relationship between this class and the instance.

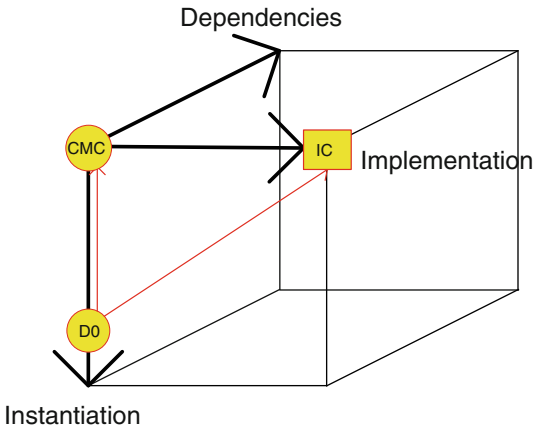


Fig. 12 The instantiation dimension: The data object (DO)

Since the CMC is represented as an object (as also shown in Fig. 13) the logical relationship can be explicitly represented.

4.4 Representing the modeled dependency dimension

In the modeled dependency dimension, we will consider a dependent CMC (dCMC), which is dependent on the CMC. In a similar way, its generated class dIMC and dIC also depend on IMC and IC, respectively, as indicated in Fig. 14.

Specialization is a special case of dependency (see also Requirement a) of Sect. 3.2). This can be used to further explain Fig. 15 by taking specialization as an example. The following rule applies for specialization. When a CMC (e.g., `Premium Customer`) is a specialization and, therefore, dependent on another CMC (i.e., `Customer`), then this specialization must be translated according to the Implementation (i.e., the Implementation `CPremiumCustomer` must be a specialization of `CCustomer`).

For the NMF, we have identified the four basic modeled dependencies possible between CMCs:

1. Generalization/specialization relationships as described above.
2. Modeled Associations which must be translated into associations of the corresponding implementation classes.
3. The "Instantiates" dependency. Sometimes instances of a CMC A can create instances of another CMC B, for example, if A is a composition of B. This dependency is explicitly represented as a relationship between CMCs. At runtime the CMC B is indeed requested to instantiate itself, such that instantiation of B is model-driven and not pre-fixed in the code.
4. Finally, there can be a "used model class" dependency. This relationship corresponds to those interactions between objects through method invocations that are not already represented in Point 2 above. In Java such dependencies will be translated to imports.

By explicitly representing these dependencies, not only Requirement (a) of Sect. 3.2 can be fulfilled, but also Requirement (d). The dependencies mentioned above can be used to trace model classes that are impacted by a replacement.

4.5 Managing the development process: the CMC object lifecycle

Another essential feature for supporting fine-grained evolution in the NMF is the lifecycle. In Nucleus each CMC Object has a so-called "lifecycle." This lifecycle defines a (principally progressive) set of states that a CMC Object goes through on its way to having an implementation class and thus being "in production," that is, such that it can be used and instantiated in the running system. This lifecycle has the following main functions:

- It guides the developer through the steps that must be taken to get the CMC into production. State transitions can be guarded by checks, for example, checking that a CMC has been correctly defined (e.g., that there are no specialization cycles). State transitions also include actions such as code generation or compilation and linking.
- It guards against potential inconsistencies between the CMC and corresponding instances. At the maturity stage of its lifecycle, a CMC has been taken into production and it will have instances in the running system. The state in

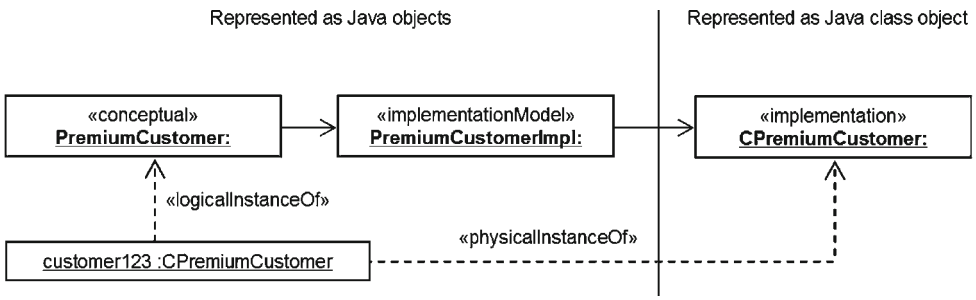


Fig. 13 Physical instantiation vs. logical instantiation in the NMF

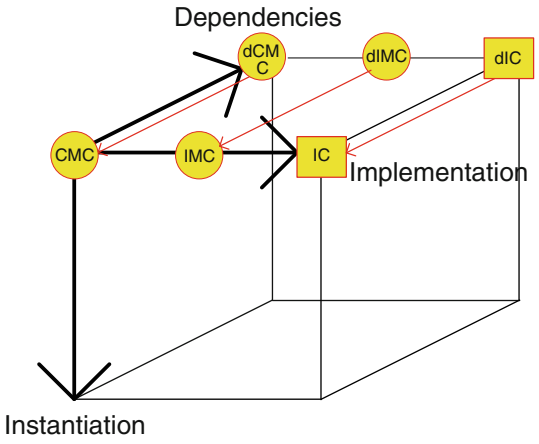


Fig. 14 The dependency dimension and the implementation dimension

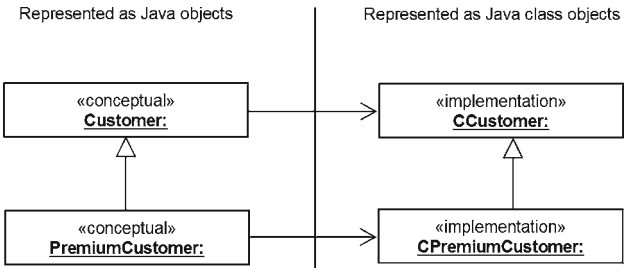


Fig. 15 The subclass relationship of CMCs is generated into the subclass relationship of the implementation classes

the lifecycle enforces the fact that such a CMC cannot be changed any more, since that will corrupt the instances. The only way to adapt the CMC is to create a new version of it (see Sect. 3.2). Instances of the old version can be moved to the new version once the new version is in production; in fact, this involves creating new versions of the instances as well. The lifecycle also includes a state in the development process in which a CMC can be tested; in this life-state the CMC can have instances, but can still be changed without requiring a new version. If the CMC is changed in this state it goes backwards in its lifecycle and its current (test) instances are removed.

- It guards against potential inconsistencies between the CMC and the implementation class. Similar to the above, the blocking of the model class in its production state ensures that the CMC cannot become inconsistent with its implementation class. During the testing phase, changes can be applied by bringing the CMC back to an earlier state of its lifecycle. In such a "backward" state transition, the current implementation class will be removed, and in a subsequent forward state transition the new implementation class will again be written.

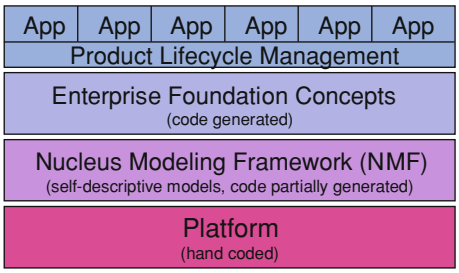


Fig. 16 Overview of Nucleus

5 Implementation of the NMF in nucleus

Figure 16 gives an overview of Nucleus in the form of a layering of subsystems that build on top of each other. This layering encompasses both runtime layering where one layer reuses another at runtime, as well as static reuse where one layer specializes another. These layers will be discussed from bottom to top. The bottom platform layer offers the functionality essential for supporting fine-grained evolution as described in this paper. In the subsequent layers functionality is added. This is done in model-driven fashion; thus the principles of this paper are applied. At the end of this section some further quantitative information will be given about Nucleus and the application of the three traceability dimensions.

One essential part of the platform layer is the persistence subsystem. The Nucleus persistence subsystem stores objects in a relational database using an object-to-relational mapping. All CMCs and IMCs (modeling information), and instances are stored here. This includes all traceability information such as the logical instantiation relationship between classes and instances. The persistence subsystem also provides essential support for the versioning (mentioned in Sect. 3.2) both of classes and of instances. The adaptability described in this paper is based on dynamic class loading in Java [39]. In the persistence subsystem a special Java class loader is used. It is used to retrieve the runtime implementation class of an instance stored in the database, such that the retrieved instance can indeed be associated with the right physical and logical class. For this purpose, the persistence subsystem stores these associations. When a CMC is replaced by means of a new version, it gets its own associated implementation class.

The platform provides support for browser-based user interfaces in which CMCs and instances alike can be edited. Figure 17 gives a screenshot of the Enterprise Application Modeler. The screenshot presents some of the many CMCs and their relationships. Note that Nucleus uses one user interface to represent and manipulate both CMCs and instances.

The Nucleus modeling framework (NMF), which is the second layer in Fig. 16, builds on top of the platform to

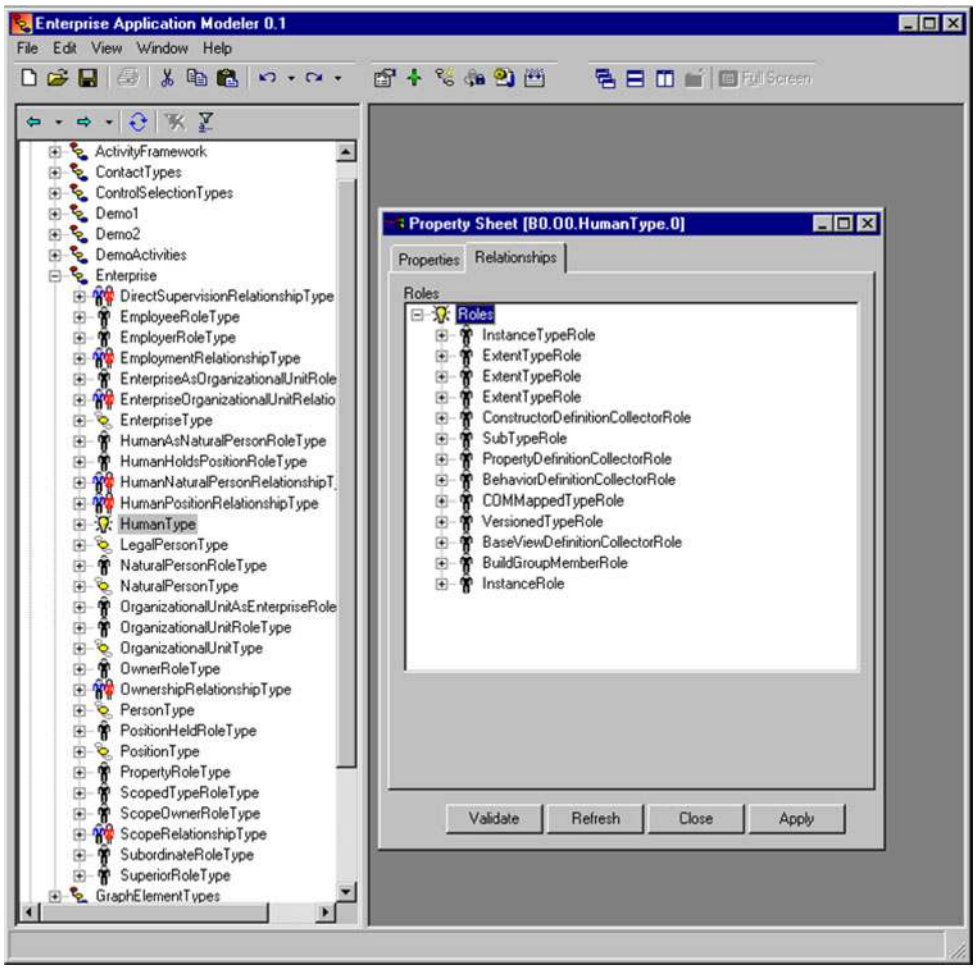


Fig. 17 Screenshot from the enterprise application modeler

define essential aspects of models in a model-driven fashion. The NMF includes a self-descriptive meta-model that defines how CMCs and IMCs are modeled in Nucleus. This meta-model thus consists of meta-CMCs and IMCs that define, for instance, the existence of a lifecycle for each CMC (see Sect. 4.5). The meta-model of Nucleus also defines how relationships between CMCs are specified. Nucleus allows for n:m association relationships where modeled relationships are represented as CMCs themselves. Due to such[5] n:m association relationships, instances can play various roles in a relationship; therefore, also roles are modeled as represented by CMCs. Details of this meta-model are beyond the scope of this paper.

Enterprise foundation concepts (EFCs) (the next layer up in Fig. 16) are those CMCs, and groups of CMCs and their relationships, that are essential for creating large Enterprise Applications. The screenshot of Fig. 17 presents a subset of the EFCs that were created. Examples of EFCs are (shown as XxType[6] in the figure):

- `Enterprise`
- `Person` (can be a legal or a natural person)
- `Human`
- `Position`

Examples of modeled relationships are (shown as YyRelationshipType[6] in the figure):

- `Human holds Position Relationship` model class
- `Position supervises Position` model class

Examples of modeled roles are (shown as ZzRoleType[6] in the figure):

- `Employee Role` model class
- `Employer Role` model class

In the figure a window is moreover shown that represents details for the CMC Human (Shown as HumanType) with its possible roles.

[5] In Nucleus the term "type" was used instead of Model Class.

Product Lifecycle[6] Management (abbreviated as e-PLM) was the first application domain of the Nucleus stack. It specializes the various EFC concepts so as to create model classes that are specific to the area of managing the lifecycle of large (physical) products.

Applications further specialize the application domain for specific customer usage. Nucleus has been successfully applied in two beta applications. One application supported the full product data management of airplane motors; this included all possible configurations and the corresponding product change management. Another supported the collaborative project management for large machines for paper production.

The mechanisms of fine-grained model-driven evolution, as subdivided into the three dimensions, have been applied as follows:

- The extension mechanism, which represents the model to model dimension, has been applied to build EFCs on top of NMF, ePLM on top of EFCs, and, once again, the definition of applications on top of ePLM include more application-specific CMCs reusing existing CMCs. From the authors' point of view, this was essential in order to create multiple real, large-scale software systems such as the two beta applications. To give an idea of the size, 218 Java classes and 262 interfaces were generated from the NMF CMCs, and IMCs. From the EFCs, 191 Java classes and 212 interfaces were generated. In the latest version of Nucleus, identifying modeled dependencies is possible, since all dependencies in Nucleus are explicitly modeled and explicitly represented, but no specific proactive support is provided for this.
- The model to implementation dimension has been used extensively in order to enable implementations to be replaced in a fine granular manner.
- Relating a model replacement (a new version of a model) to a corresponding upgrade of the instances, again in terms of new versions of these instances, has become possible, but is not widely used.

In spite of this, the Nucleus approach has never lived up to its full promise. The project was discontinued 7 years after it began. As a result of this discontinuation, the software is no longer available. There are several reasons for this discontinuation: Building the platform itself was a large undertaking which took about 100 man-years of effort; the incremental modeling of Nucleus could only function in a complete platform, with the promise that it might lead to an ever-accelerating development speed. But once the platform was able to be used in this way, the project was bought by another organization that was not able to integrate it into its portfolio.

As a result of this effort, the Nucleus product consists, in a large part, of a proprietary platform, with a corresponding burden for maintenance. This is another reason why it has not been continued. Nucleus has been mentioned by Forrester [12].[7]

6 Discussion

The principles introduced in this paper raise various issues:

1. Strict meta-modeling [40] is often regarded as an important principle of MDD. It allows for only one way of instantiation. How can the fact that an instance has multiple ways of instantiation, a logical way and a physical way, relate to this?
2. One of the important targets of MDD in general and the MDA in particular is to allow for implementation-independent modeling, enabling models to be mapped to different implementation platforms. Given that this architecture requires that the modeling tool and the runtime tool be implemented in one and the same runtime platform, can implementation independence still be maintained?
3. Can the requirements of Sect. 3.2 be fulfilled and to what extent?

These issues will be discussed in more detail below.

6.1 Strict meta-modeling

In [40] Atkinson and Kuhne define strict meta-modeling as follows:

"*In an n-level modeling architecture,* $M0$, $M1 \ldots Mn-1$, *every element of an Mm-level model must be an instance-of exactly one element of an* $Mm+1$*-level model, for all* $m < n-1$, *and any relationship other than the instance-of relationship between two elements X and Y implies that level(X)* $=$ *level(Y).*"

Due to the two ways of instantiation, the approach presented in this paper seems to depart from that approach. However, in the paper by Atkinson and Kühne [11] it is shown that a logical instantiation and a physical instantiation represent different (basically orthogonal) concerns, since any logical model class can theoretically still have various different possible realizations. Any instance can have both a logical and a

[6] The term lifecycle should not be confused with the lifecycle of model class objects introduced earlier.

[7] It is mentioned under the name "Xebic," which is the spin-off company of the research project that started the development.

physical class. Thus, the strict meta-modeling principle *does not cover* the fact that different modeling elements can represent different aspects that allow for multiple types. A relevant clue for this is that the classes will complement each other and will not contradict. A generated class is the refinement of the other.

It may be considered even more critical that in the NMF the levels are mixed: model classes are all represented as objects and thus model classes of different meta-levels all live in the same environment. However, this situation does not fundamentally break the rule: each and every model class can be related to one level such that the strict situation can be reached again.

6.2 Implementation independence

As indicated by the MDA user's guide [1], one of the major targets is to enable PIM-like models to be mapped to different implementation platforms. Thus, implementation independence is one important goal of the MDA. Evolution in the NMF is only supported for the preferred platform, that is, when code generation is done for classes and instances that live in the same platform as the (as objects represented) model classes. This has been shown to be relevant for the NMF principles described in Sects. 4.2 and 4.3. Thus, NMF seems to jeopardize implementation independence.

The following counterpoints can be made to this argument:

1. The NMF environment itself can be (and has been) ported to different execution platforms. Thus, fine-grained evolution can be supported on all execution platforms on which the NMF has been implemented.[8]
2. Generation to platforms in which NMF has not been implemented is still possible; for such platforms the feature of fine-grained evolution is merely lost.
3. Another possibility of the NMF is to use special "proxy objects" that can execute method requests on behalf of normal objects and to translate these requests to some other runtime environment, as, for example, translating requests to webservice invocations.

Still, it is true that in this respect the NMF is closer to interpreted approaches and has the corresponding portability problems, as mentioned in Sect. 2.2, as "normal" generative approaches do. It turns out that there is a trade-off between requirements with respect to fine-grained evolution and requirements with respect to ease of portability.

[8] The NMF implementation has been ported from a Java environment to a C# environment in half a man year.

6.3 Fulfilling the requirements and improving on interpreted approaches

The NMF fulfills the requirements of Sect. 3.2 in the following ways:

- Requirement a): Section 4.2 describes how a model class for which the development process has been finished refers to its class. Sect. 4.4 describes how modeled dependencies are translated to corresponding relationships in the implementation. Together these mechanisms ensure that the class of a new or replaced model class will be correctly hooked up with the other pre-existing classes, thus with the implementation of the rest of the system.
- Requirement b): Through the mechanism described in Sect. 4.1, a model class is represented in runtime as an object; in fact, the modeling environment and runtime environment are integrated, similar to interpreted approaches. Moreover, as described in Sect. 4.2, the model class will also refer to its implementation once the development process is finished. As a result, a newly introduced and developed model class is a factory object which can be instantiated.
- Requirement c): Through the runtime representation of model classes as objects and the explicit representation of the relationship between a model-class object and its instances (the logical instance of relationship) as described in Sect. 4.3, dependent data objects, i.e. instances, can be traced.
- Requirement d): Model classes are explicitly represented as objects; relationships between model classes are also explicitly represented. Thus, tracing dependencies is supported in principle. This is, however, clearly not an outstanding property of the NMF. Any modeling environment will use some form of explicit internal representation of modeling elements and support some form of traceability in this dimension. Of course, such traceability requires representing dependencies (e.g., invocation relationships between model classes) as explicitly as possible.

7 Related work

7.1 Integrating component-based development with model-based development

In their paper, Tongren et. al. [3] describe how model-driven development (MDD) and component-based development (CBD) are complementary and need each other. One example of such an integration is where components have been developed through MDD, but are glued together with CBD. On the other hand, gluing together components requires models. According to them, a complete integration must still be

achieved. The NMF provides a contribution here, since it supports the model-driven maintenance of a complete system, mapping each business type of the NMF to one small component (see also [41]), but other mappings are also possible, see [22]. Interestingly, models in the NMF use a similar versioning to that of components [42] and thus reflect the versioning of components.

7.2 Interpreted approaches

Interpreted model-driven approaches are described, among others, by Riehle et. al. [10], and Atkinson and Kühne [11, 32]. As described in Sects. 2.1 and 2.2, interpreted model-driven approaches support direct adaptations of the models, leading to corresponding direct adaptation of the runtime system and are, in this respect, closer to adaptive MDD than generative approaches. Still, as also described in Sect. 2.2, these approaches do not support evolution very well, due to the resulting brittleness where all kinds of dependencies are easily corrupted. The NMF addresses such brittleness as a result of its explicit versioning and its use of a lifecycle that blocks direct adaptation once instances have been created.

Still, the NMF reuses various concepts from these approaches:

- The integration between modeling environment and runtime environment, such that the modeled system maintains its own models;
- The representation of partial models as objects, as a consequence;
- The distinction between logical and physical instantiation.

Thus, the NMF can be seen as a hybrid approach, combining essential aspects of these interpretive approaches with code generation. The innovation of the NMF with respect to these concepts is that it generates the class of a model class "dynamically" instead of using a pre-fixed set of classes that implement the interpreter.

Similar to the work of Riehle, as well as Atkinson and Kühne, our work also has roots in original work on reflection, such as that by Pattie Maes [43]. Reflection is the ability of a system to reason about and act upon itself by means of a powerful self-representation. Clearly our approach is reflective in this sense. Again, however, the reflective approaches we know of are interpretive by nature.

7.3 Model-driven program transformation

Gray et al. [44] describe another approach to supporting model-driven evolution, which they call model-driven program translation. In this approach, model evolution translates to changes in code transformation software. The code transformation software processes the original source code. This approach allows deltas to be mapped to deltas in the final code. Similar to the approach presented in this paper, certain model elements correspond to certain components in the source. These components are much bigger than just single classes, as presented in this paper. Their approach does not cover the impact on instances, however, which is a relevant part of the presented approach. Moreover, adding new components does seem to be integrated in the transformational approach, since it processes existing components.

7.4 The type object pattern

There is a direct relationship between this work and the so-called type object pattern [18,45]. The type object pattern is also known as the adaptive object pattern. The integration between modeling environment and runtime environment is based on this pattern.

The work of Razavi et al. [46] is especially relevant, as they have integrated the type object pattern into well-known languages such as CLOS that support meta classes. This approach may well improve on some of the disadvantages of interpreted approaches such as those mentioned in Sect. 2.2, especially with respect to non-functional aspects such as performance and storage. However, the potential danger of ad-hoc changes, and the portability problem seem to remain.

7.5 The virtual machine principle

Fine-grained evolution is not a new feature. In the Java virtual machine especially, which is based on the Smalltalk virtual machine, evolution is supported [47]. As in our approach, in these approaches one can see that class relationships are explicitly represented and can be mapped to byte-code relationships.

7.6 Formalization of incremental development

More fundamentally, this work can be seen as an application of the theory on incremental computation [48], and the support for that through function caching [49]. Roughly stated, the approach of [49] is to allow results of functions—in this case the transformation function from source model to target model or implementation—to be cached, so that new inputs can be incrementally translated to the output. In this case, the output of the translation of the new type is incrementally added to the "cached" output of the previous translation.

7.7 Impact calculation

Work has been done on calculating both model to model impact [28,29] (see also [50]), as well as calculating data

impact [51]. In fact, that kind of work is complementary to the work presented in this paper.

7.8 Generative programming and product lines

The approach of Czarnecki [52,53] which is pretty much related to Generative Programming, is also relevant here. The idea is to describe a product line instead of a single product and provide code/instructions on how to assemble the pieces. It does not, however, handle the issue of evolution. A similar approach is GenVoca [54], which is an advanced architecture providing lots of support for reuse and suchlike. It is based on mathematics, and specifically geared towards software changes. Nevertheless, all changes happen at the model level and before the generation, everything is known.

7.9 Other approaches to model-driven evolution

Other authors have done work on supporting evolution through MDD. Favre and NGuyen [55] theoretically discuss and model all possible evolution steps in MDD. However, their work is not explicitly directed at incorporating incremental evolvability in the development process. Hearnden et al. [56] also analyze dependencies in MDD. Again, a direct link with tooling is not made. Birken [57,58] describes patterns for enabling evolution, especially those based on traceability. This approach is directed at improving the design and design practices so as to improve evolvability; it is not directed so much at offering technical solutions as part of the modeling architecture itself.

7.10 Adaptive systems

Adaptive Systems is a wide area [57,58] that is related to this paper. Adaptive Systems are defined as Systems that can be adapted to accommodate resource variability, changing user-needs and system faults [59]. Specific adaptive approaches lie in the area of architectural adaptability [59,60] where (architectural) models and generative approaches play a role. Our approach combines generation with (instantiable) class models, which is not covered by such approaches. Other adaptation approaches enable adding new software "patches", e.g., function or function versions [61,62], but these are changes at the code level, not at the model level; moreover, the addition of properties and corresponding impact on instances is not treated. In [57,58] an overview of mechanisms is given for run-time (self-) adaptive systems. On basis of a set of fundamental topics a taxonomy of mechanisms is provided. Of the described mechanisms, only Aspect-oriented Programming is based on generative techniques; however, the combination between generation and (model-driven) abstraction is not mentioned in the paper.

8 Conclusion and further work

Evolving large-scale systems is a complex undertaking due to the many dependencies [25]. It can be difficult, therefore, to handle the impact of a change. Especially for large-scale enterprise systems, evolvability is a very important quality, due to the great dynamicity of the enterprises that must be supported by these systems [6]. Model-driven approaches to software development should, in principle, be very suitable for supporting such evolvability, since models can precisely and formally describe dependencies [25].

Standard environments for MDD, however, have a strong separation between modeling environment and runtime environment [10]. As a result, they are insufficiently directed toward evolving very large-scale systems. Letting each model change lead to a complete regeneration of a system is not feasible and sometimes not even possible (i.e., reusing a modeled component where code is not available). Regenerating only smaller parts of a system (components or frameworks) in isolation can cancel out the advantages of using MDD, since the relationships with the rest of the system are not taken into account. Moreover, due to the separation between modeling environment and runtime environment, the impact of model changes on existing data is difficult to support. However, these data are an essential part of the system, especially in enterprise systems.

This paper describes an integrated modeling-runtime environment that forms the basis for enabling fine-grained evolution of large-scale enterprise systems using generative techniques. This integration means that the runtime system maintains its own models and can, therefore, be locally adapted, without missing impact information. The integration between modeling and runtime, moreover, ensures that all dimensions of possible impacts of a change can be traced: (1) between models and dependent models; (2) between models and implementation; and (3) between models and instances, the data. The integration of modeling and runtime comes at a cost, however, as it is traded off against a diminished support for platform independence. Fine-grained evolution is only supported for the "preferred platform" in which the integrated environment is realized, similar to interpretive approaches that are more strongly bound to those platforms for which an interpreter is available.

The NMF is a hybrid approach that combines code generation with certain ideas from interpreted approaches such as those described by Riehle et al. [10], and Atkinson and Kühne [11,32]. In comparison with these approaches, it adds the dimension of implementation and generation, and removes the brittleness that comes with the possibility of being able to directly implement a change in the model. The NMF embodies the main concepts of a large research project called Nucleus. Since the NMF was originally developed in a commercial setting, many other aspects surrounding the NMF

must still be reported on, and this, therefore, constitutes future work. Some examples of this are the evolution of database schemas and the evolution of user interface aspects. The NMF does not explicitly support the removal of model classes. However, it does allow for replacing groups of model classes with corresponding new versions and, on top of this, a mechanism may yet be offered to restructure the data correspondingly. This will also be the subject of further study.

Acknowledgments The authors wish to thank the Baan Nucleus, and their successors from the Xebic team, for their work on Nucleus. The authors would also like thank all those who supported the writing of this paper, such as Marieke Vreugdenhil, Douwe Postmus, Ashwin Ittoo and many others.

Open Access This article is distributed under the terms of the Creative Commons Attribution Noncommercial License which permits any noncommercial use, distribution, and reproduction in any medium, provided the original author(s) and source are credited.

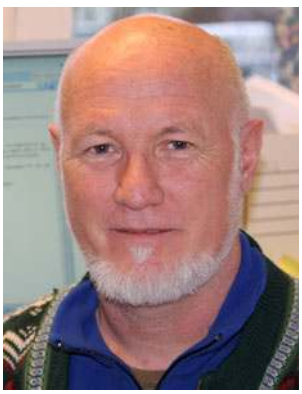
References

1. Mukerji, J., Miller, J. (eds.): MDA Guide Version 1.0.1. Technical report, Object Management Group. http://www.omg.org/docs/omg/03-06-01.pdf (2003)
2. Kleppe, A., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley, Reading (2003)
3. Torngren, M., Chen, D., Crnkovic, I.: Component-based vs. model-based development: a comparison in the context of vehicular embedded systems. In: Proceedings of the 31st EUROMICRO Conference on Software Engineering and Advanced Applications, Porto, Portugal, pp. 432–441 (2005)
4. OMG MDA success stories. http://www.omg.org/mda/products_success.htm
5. Knoll, K., Jarvenpaa, S.L.: Information technology alignment or 'fit' in highly turbulent environments: the concept of flexibility. In: Proceedings of the 1994 Computer Personnel Research Conference on Reinventing IS: Managing Information Technology in Changing Organizations, Alexandria, Virginia, USA, pp. 1–14 (1994)
6. Lehman, M.M., Ramil, J.F.: Evolution in software and related areas. In: Proceedings of the 4th International Workshop on Principles of Software Evolution, Vienna, Austria, pp. 1–16 (2001)
7. Papazoglou, M.P., van den Heuvel, W.J.: Service oriented architectures: approaches, technologies and research issues. VLDB J. **16**(3), 389–415 (2007)
8. White, S.: Business process modeling notation (BPMN). Version 1.0—May 3, 2004. http://BPMI.org (2004)
9. Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., Weerawarana, S.: Business process execution language for web services, version 1.1. BEA Systems, IBM Corporation, Microsoft Corporation, SAP AG, Siebel Systems. (2003)
10. Riehle, D., Fraleigh, S., Bucka-Lassen, D., Omorogbe, N.: The architecture of a UML virtual machine. In: Proceedings of the 2001 Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01), Tampa Bay, Florida, USA, pp. 327–341 (2001)
11. Atkinson, C., Kühne, T.: Rearchitecting the UML infrastructure. ACM Trans. Model. Comput. Simul. **12**(4), 290–321 (2002)
12. Homs, C., Metcalfe, D., Nordan, M. M., Radjou, N.: Troubled invensys: Dispose of fading baan, In: Forrester brief, August 21, (2002)
13. Mellor, S.J., Balcer, M.J.: Executable UML: A Foundation for Model-Driven Architecture. Addison-Wesley, Reading (2002)
14. Visser, E.: Meta-programming with concrete object syntax. In: Batory, D., Consel, C., Taha, W. (eds.) Generative Programming and Component Engineering (GPCE 2002), LNCS 2487, pp. 299–315. Springer, Pittsburgh (2002)
15. Aho, A., Lam, M., Sethi, R., Ullman, J.: Compilers, Principles, Techniques and Tools, 2nd edn. Pearson Education. Addison Wesley, Reading (2007)
16. Watt, D., Brown, D.: Programming Language Processors in Java. Prentice-Hall, Englewood Cliffs (2000)
17. Benjamin, C.: Pierce, Types and Programming Languages. MIT Press, Cambridge (2002)
18. Yoder, J.W., Balaguer, F., Johnson, R.: Architecture and design of adaptive object-models. ACM SIGPLAN Notices **36**(12), 50–60 (2001)
19. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Model Language Reference Manual, 2nd edn. Pearson Education Inc, Upper Saddle River (2005)
20. Meyer, B.: Object-Oriented Software Construction, 1st edn. Prentice-Hall, Englewood Cliffs (1988)
21. Unified modeling language: Superstructure, version 2.1.2. http://www.omg.org/cgi-bin/doc?formal/07-11-02
22. Pernici, B., Mecella, M., Batini, C.: Conceptual modeling and software components reuse: towards the unification. In: Solvberg, A., Brinkkemper, S., Lindencrona, E. (eds.) Information Systems Engineering: State of the Art and Research Themes, pp. 209–220. Springer, London (2000)
23. Maciaszek, L.A.: Requirements Analysis and System Design: Developing Information Systems with UML. Addison-Wesley, Reading (2005)
24. Ducasse, S., Girba, T., Favre, J.M.: Modeling software evolution by treating history as a first class entity. In: Proceedings of the Workshop on Software Evolution Through Transformation (SETra 2004), Rome, Italy, pp. 75–86 (2004)
25. France, R., Bieman, J.M.: Multi-view software evolution: a UML-based framework for evolving object-oriented software. In: Proceedings of the 17th IEEE International Conference on Software Maintenance (ICSM'01), Florence, Italy, pp. 386–395 (2001)
26. Mens, T., Tourwé, T.: A Survey of software refactoring. IEEE Trans. Softw. Eng. **30**(2), 126–139 (2004)
27. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., Irwin, J.: Aspect-oriented programming. ECOOP **97**, pp. 220–242 (1997)
28. Mens, T., D'Hondt, T.: Automating support for software evolution in UML. Autom. Softw. Eng. **7**(1), 39–59 (2000)
29. Steyaert, P., Lucas, C., Mens, K., D'Hondt, T.: Reuse contracts: managing the evolution of reusable assets. In: Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'96), San Jose, California, USA, pp. 268–285 (1996)
30. Frankel, D.S.: Model Driven Architecture: Applying MDA to Enterprise Computing. Wiley, New York (2003)
31. Cook, S.: Domain-specific modeling and model driven architecture. MDA J. pp. 1–10 (2004)
32. Atkinson, C., Kühne, T.: Model-driven development: a metamodeling foundation. IEEE Softw. **20**(5), 36–41 (2003)
33. Stefik, M., Bobrow, D.: Object-oriented programming: themes and variations.. AI Mag. **6**(4), 40–62 (1986)
34. Ivan, K., Jean, B., Frédéric J., Patrick, V.: Model-based DSL frameworks, OOPSLA companion, pp. 602–616 (2006)
35. Henderson-Sellers, B.: The use of subtype and stereotypes in the UML model. J. Database Manag. **13**(2), 43–50 (2002)
36. Atkinson, C., Kühne, T.: A generalized notion of platforms for model driven development. In: Beydeda, S., Book, M., Gruhn, V.

(eds.) Model-Driven Software Development, pp. 119–136. Springer, Berlin (2005)
37. D'Souza, D., Sane, A., Birchenough, A.: First class extensibility for UML—packaging of profiles, stereotypes, patterns. In: Proceedings of the 2nd International Conference on the Unified Modeling Language (UML'99), Fort Collins, USA, pp. 265–277 (1999)
38. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1995)
39. Liang, S., Bracha, G.: Dynamic class loading in the Java Virtual Machine. OOPSLA, pp. 36–44 (1998)
40. Atkinson, C., Kühne, T.: Profiles in a strict metamodeling framework. Sci. Comput. Program. **44**(1), 5–22 (2002)
41. Meijler, T.D., Kruithof, G.H., van Beest, N.S.: Top down versus bottom up in service-oriented integration: An MDA-based solution for minimizing technology coupling. In: Proceedings of the 4th International Conference in Service-Oriented Computing, Chicago, IL, USA, pp. 484–489 (2006)
42. Stuckenholz, A.: Component evolution and versioning state of the art. ACM SIGSOFT Softw. Eng. Notes **30**(1), 1–13 (2005)
43. Maes, P.: Concepts and experiments in computational reflection. ACM SIGPLAN Notices **22**(12), 147–155 (1987)
44. Gray, J.G., Zhang, J., Lin, Y., Roychoudhury, S., Wu, H., Sudarsan, R., Gokhale, A.S., Neema, S. Shi, F., Bapty, T.: Model-driven program transformation of a large avionics framework. Generative Programming and Component Engineering (GPCE 2004) LNCS. **32**(86), 361–378 (2004)
45. Yoder, J.W., Johnson, R.: The adaptive object-model architectural style. In: Proceedings of the IFIP 17th World Computer Congress—TC2 Stream. 3rd IEEE/IFIP Conference on Software Architecture: System Design, Development and Maintenance, Montreal, Quebec, Canada, pp. 3–27 (2002)
46. Razavi, R., Bouraqadi, N., Yoder, J.W., Perrot, J.F., Johnson, R.: Language support for adaptive object-models using metaclasses. Comput. Lang. Syst. Stru. **31**(3–4), 199–218 (2005)
47. Goldberg, A., Robson, D.: Smalltalk-80: The Language and Its Implementation. Addison-Wesley, Reading (1983)
48. Sundaresh, R.S., Hudak, P.: A theory of incremental computation and its application. In: Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Orlando, Florida, USA, pp. 1–13 (1991)
49. Pugh, W., Teitelbaum, T.: Incremental computation via function caching. In: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming languages, Austin, Texas, USA, pp. 315–328 (1989)
50. Sprinkle, J., Karsai, G.: A domain-specific visual language for domain model evolution. J. Vis. Lang. Comput. **15**(3–4), 291–307 (2004)
51. Monk, S., Sommerville, I.: Schema evolution in OODBs using class versioning. ACM SIGMOD Record. **22**(3), 16–22 (1993)
52. Krzysztof, C., Michal, A., Chang, H., Peter, K., Sean, L., Krzysztof, P.: Model-driven software product lines. OOPSLA Companion: 126–127 (2005)
53. Krzysztof, C.: Software reuse and evolution with generative techniques. Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering (ASE'07), p. 575 (2007)
54. Don, S.B., Clay, J., Bob, M., Dale, V.H.: Achieving extensibility through product-lines and domain-specific languages: a case study. ICSR 2000, LNCS. **18**(44), 117–136 (2000)
55. Favre, J.M., Nguyen, T.: Towards a megamodel to model software evolution through software transformation. Electron. Notes Theor. Comput. Sci. **127**(3), 59–74 (2005)
56. Hearnden, D., Bailes, P., Lawley, M., Raymond, K.: Automating software evolution. In: Proceedings of the 7th International Workshop on Principles of Software Evolution, Kyoto, Japan, pp. 95–100 (2004)
57. Berkem, B.: How to increase your business reactivity with UML/MDA? J. Obj. Technol. **2**(6), 117–138 (2003)
58. McKinley, P.K., Masoud, S., Kasten, E.P., Cheng, B.H.C.: Composing adaptive software. IEEE Comput. **37**(4), 56–64 (2004)
59. Garlan, D., Cheng, S.W., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. IEEE Comput. **37**(10), 46–54 (2004)
60. Oreizy, P., Gorlick, M.M., Taylor, R. N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici A., Rosenblum, D.S., Wolf, A. L.: An architecture-based approach to self-adaptive software. IEEE Intell. Syst. 14(3), 54–62 (1999)
61. Sadjadi, S.M., Trigoso, F.: Trap Net: a realization of transparent shaping in net. Int. J. Softw. Eng. Knowl. Eng. **19**(4), 507–528 (2009)
62. Stachour, P., Collier-Brown, D.: You don't know Jack about software maintenance.Commun. ACM **52**(11), (2009)

Author Biographies

Theo Dirk Meijler is currently senior researcher at SAP. He studied at the Technical University of Delft, The Netherlands. He received his Ph.D. at the Erasmus University Rotterdam. He has worked as post-doc and lecturer at academic institutions in Switzerland (Berne) and the Netherlands (Groningen) and as senior engineer at the research department of Baan and Xebic. His main interests are adaptable service-based systems, currently focusing on run-time adaptable systems for workflow, business processes, and choreography-based coordination.



Jan Pettersen Nytun has since 1984 been holding several positions at academic and industrial institutions in Norway. He studied computer science and mathematics at the University of Oslo, and received his M.Sc. in computer science (1990). He has been employed at UiA since 1994 and he is currently working on a Ph.D. in computer science at UiO. The Ph.D. work focus on modeling of consistency and multi-model architectures. His research interests and competence include object-orientation, modeling, and programming languages.

Andreas Prinz was appointed head of ICT at UiA in 2007. He studied mathematics and computer science at the Humboldt-University in Berlin, Germany, and received his M.Sc. in mathematics (1988) and Ph.D. (1990) in computer science there. From 1990 to 2007 he had several positions at academic and industrial institutions in Germany, Australia and Norway. His research interests and competence include systems engineering with particular focus on modeling, languages, and formal methods. Prof. Prinz has worked in several projects dealing with the development of modern ICT systems using advanced technology.

Hans Wortmann (1950) is full professor in Information Management at the Faculty of Organization and Management within the University of Groningen (RuG). His special field of interest is in enterprise information systems. He is chairing a platform in The Netherlands om Software-as-a-Service. He is Editor-in-Chief of the applied scientific journal *Computers in Industry.* Before joining RuG, Hans Wortmann was employed at Baan Company, a leading vendor of standard enterprise software, as Vice President in charge of R&D. Before joining Baan, Hans served as a full professor in industrial information systems at Eindhoven University of Technology (TUE). In this role, Hans gained much experience with enterprise modeling and enterprise systems. He advised many companies in various industrial branches on selection and implementation of information systems in enterprises. He received his Ph.D. in Engineering at TUE in 1981.