Meta-Model and Model Co-evolution within the 3D Software Space

Jean-Marie Favre

Adele Team, Laboratoire LSR-IMAG
University of Grenoble, France
http://www-adele.imag.fr/~jmfavre

Abstract

Software evolution-in-the-large is a challenging issue. While most research work concentrates on the evolution of "programs", large scale software evolution should be driven by much higher levels of abstraction. Software architecture is an example of such abstraction. The notion of co-evolution between architecture and implementation has been identified and studied recently. This paper claims that other abstraction dimensions should also be taken into account, leading to what we call the 3D software space. This conceptual framework is used to reason about evolution-in-the-large phenomena occurring in industry. The meta dimension, which constitutes the core of the MDA approach, is considered as fundamental. This paper makes the distinction between appliware and metaware and put the lights on meta-model and model co-evolution. Conversely to the MDA approach which makes the implicit assumption that meta-models are neat, stable and standardized, in this paper meta-models are considered as complex evolving software artefacts that are most often recovered from existing metaware tools rather than engineered from scratch. In fact, we identified the notion of meta-model and model co-evolution in the context of the evolution of a multi-million LOC component-based software developed by one of the largest software companies in Europe.

1. Introduction

Understanding very large software products is a major issue. Understanding their *evolution* is even more difficult since many factors influence software evolution [1][2]. This paper concentrates on *evolution-in-the-large* which is quite different from *evolution-in-the-small*, that is evolution of small programs over rather short periods of time (a few months or years). Evolution-in-the-large is about the evolution of multi-million LOC software over decades.

Evolution-in-the-large is indeed a very complex issue. Considering software evolution at the level of statements and functions is clearly not enough. A much higher level of abstraction is required.

1.1. Architecture/Implementation co-evolution

Software architecture shoutld clearly play a central role in the evolution since it provides a abstraction. However, making explicit the architecture of software is not easy in practice. Architectural Description Languages (ADLs) failed to find their path to industry, in part because of their poor support for software evolution. Software industry is still code-centric. Most of the time the architecture is implicit. To cope with this problem, a increasing amount of research work focuses on architecture recovery and architecture evolution (e.g. [3][4][5][6][7]). Recently the concept of *architecture and implementation co-evolution* has been identified by various authors (e.g.[9][10][11]). Architecture and implementation are two levels of abstraction. They are both subject to evolution. Since they are linked they should ideally evolved in a synchronized way to avoid the so called *architectural drift* and *architectural erosion*. Maintaining some kind of architectural description is useful to ease software understanding, but another important idea is that the architectural description should allow to *drive* or at least to *constrain* the evolution of implementation.

Figure 1 provides a very intuitive view of the relationship between the two abstraction levels. Modifying an entity at a level of abstraction can both have an impact at this same level, but also at the lower (or higher) level of abstraction. In fact *whenever two entities are linked by a relation, changing one entity may have some impact on the other one*.

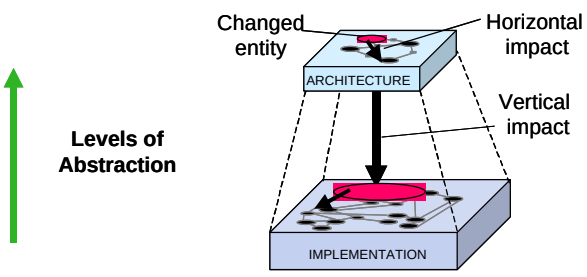


Figure 1. Vertical vs. horizontal impacts

Horizontal impacts refer to impacts within a given abstraction level (i.e. modifying a function may imply to upgrade other functions). By contrast, *vertical impacts* cross abstraction levels. For instance modifying a function may have an *upwards impact* on an architectural component. Removing a dependency between two components may have many *downwards impacts* on implementation entities. The nature of the impact obviously depends on the nature of the entities and the nature of the relation. The same is true for the action to be taken after such an impact is detected.

Horizontal consistency must usually be ensured. For example updating the impacted functions is usually considered of paramount importance to ensure a consistent behaviour of the implementation. *Vertical consistency* could be much more loose leading to a large range of co-evolution policies. For instance, upgrade could sometimes be deferred, taking the risk of a temporary inconsistency and deviation [9]. With no suitable policy, these inconsistencies usually lead to irremediable erosion and the very common situations where architectural artefacts are no longer updated. In fact, the horizontal dimension has been studied for long leading for instance to research on impact analysis either at the implementation level or at the architectural level (e.g. [8]). The term co-evolution is usually used vertically when the evolution of two levels of abstraction can be asynchronous.

1.2. Co-evolution along other dimensions

We discovered over the last years the existence of similar phenomena along other abstraction dimensions. Co-evolution is indeed a very common. This paper introduces two other abstract dimensions and structure the set of software artefact as a *3D software space*. This *conceptual framework* is very useful in the systematic identification of co-evolution processes.

In particular the main objective of this paper is to put the light on meta-model and model co-evolution. Though this phenomenon occurs along the meta-dimension popularized by the UML and MDA standards [12][13][14], the concepts presented in this paper are by no means restricted to software developed using modern techniques such as Model Driven Engineering (MDE) [15][16][17][19] (In this paper MDA refers to the OMG standard while MDE refer to the approach which is more general). This paper shows that meta-model and model co-evolution actually occurs with current and legacy industrial practices.

In fact, the MDA approach assumes that meta-models are neat, stable and standardized. In this paper on the contrary meta-models are considered as complex evolving software artefacts that are most often recovered from existing tools rather than engineered from scratch. Simply put, while the MDA and meta-related technologies are typically oriented towards forward engineering, this paper considers meta-models in the context of reverse engineering.

1.3. Background

In fact, the concepts presented in this paper results from our experience in various industrial settings. In particular, we first identified the meta-model and model co-evolution phenomenon in the context of a collaboration with Dassault Systèmes (DS). DS is the world leader in CAD/CAM and one of the largest software company in Europe. Our collaboration with this company lasted 7 years. During this period we dealt with many issues related with software evolution including configuration management, software architecture and reverse engineering [20][21]. We gained a lot of expertise about evolution-in-the-large. In fact, DS faces a wide range of issues related with very large scale software evolution. More than 1200 software developers work at the same time on the same software product leading to tremendous requirements in configuration management [20]. DS evolves a huge software, CATIA, which is made of more than 70 000 classes, 800 frameworks, and 3000 DLLs. This leads to tremendous requirements on software architecture [22][23]. In fact, Dassault Systèmes is with Microsoft one of the pioneer of component-based software development. In the mid 90's DS started to design and develop an in-house component-technology called the OM and at the same time this technology was used to develop CATIA components [21]. It will be shown in this paper that this is in fact a typical example of meta-model and model co-evolution.

The rest of the paper is structured as following. In Section 2 a simplified explanation of what is meta-model and model co-evolution is provided. Section 3 gives an overview of the conceptual framework referred as the "3D software space". The first dimension, called the meta-dimension, is presented in section 4. Section 5 introduces the "product engineering dimension". Section 6 describes the third dimension, the "representation dimension". Section 7 shows how evolution interacts with this 3D space. Section 8 gives examples of observable meta-model/model co-evolution phenomenon. Finally section 9 concludes the paper.

2. Language/Program/Tool co-evolution

Meta-related notions could be difficult to grasp at the first sight, especially when applied in complex industrial contexts. Before to introduce the 3D software space in a systematic way, let us introduce the issue in terms of much more narrow but much more intuitive concepts. For the sake

of clarity, the illustrating problem is based on well-known programming-in-the-small concepts. Let us consider three kind of entities: *programs*, (programming) *languages*, and (language-dependent) *tools* (e.g. compilers). Three kinds of relation can be considered: (1) language/program, (2) tool/ program, (3) tool/language. All these relations leads to co-evolution issues as suggested below.

2.1. Language / program co-evolution

A program is closely linked with the language it is written in. It is well known that a change in the language could have a strong (downwards) impact on programs. This leads to a wide range of upgrade and migration strategies. When a new version of the language is made available, developers have first to determine which programs are impacted by the language modification. They could then decided to upgrade impacted programs to ensure consistency with the new language. Alternatively they could delay the changes and continue to use the old language version. They might to that for impacted programs while using the new version to develop new programs. This common situation reveals *language and program co-evolution*. This phenomenon is usually not made explicit.

2.2. Tool / program co-evolution

Language dependent tools such as interpreters, compilers also have a great influence on programs. The availability of such *primary tools* is of fundamental importance in practice. *Secondary tools* such as documentation generators, metric and profiling tools are also very appreciated in industrial settings, in particular in the context of quality insurance processes. Developers may have to adapt their programs to use a particular tool. This could be to take advantages of a feature (e.g. adding tags in comments to use a documentation generator like javadoc). Sometimes this is to avoid a bug in the tool (e.g. removing the use of C++ templates in a program because the compiler on a given platform do not handle it properly). Tool evolution leads to *tool and program co-evolution* issues.

2.3. Language / tool co-evolution.

Languages are abstractions. Tools are concrete implementations supporting these languages. A change in a language specification could have many impacts on many tools. This leads to *language and tool co-evolution*. Upgrading primary tools such as compiler and interpreters is usually done first to get synchronized with the language. By contrast, the modification of secondary tools such as browsers are often delayed. Deviation from the language specification is common for such tools.

2.4. Discussion

Summing up, programs, languages and tools are linked by three kinds of relation. Each relation give rises to co-evolution issues. At this point the reader might not be convinced by the relevance of these issues. A few observations must be made to relate the discussion to the context of evolution-in-the-large.

One might argue that changes in languages and tools are seldom when compared to changes in programs. This is quite true but remember that the time scale considered in this paper is expressed in terms of years or decades, not weeks or months. Everything evolve in large companies. Software architecture evolve. Tools evolve. Languages evolve. While small projects apply versionning concepts to programs, large scale projects also deal with *language versionning* and *tool versionning*. Languages and tools are consider as actual part of the software, which is very true.

It is also very important to stress that while the term “language” might evoke to researchers a neat, standardized and stable thing, “real-life” is industry is often quite different. To a large extend, today software industry largely relies on many ill-defined, proprietary and unstable languages. The same is true for tools. The goal of this paper is to model industrial practices as they are.

Taking into account legacy software and legacy practices is an important requirements. In the early decades of computer science, many large companies developed in-house programming languages and made them evolved incrementally while developing programs at the same time. These are real-life language/program co-evolution scenarios in which language evolution is driven by the problems encountered in developing programs.

Note that in this context the language remains most of the time implicit; there is no explicit description of the language. As reported in [24], the exact grammar of programming languages such as COBOL variants is often unknown and has to be recovered from tools. This leads to grammar reverse engineering [24]. Many legacy and proprietary languages have actually evolved mostly through patches in compilers or interpreters to add or remove special features. Many language definitions, if ever existed, deviated from tool evolution and became inconsistent. This is *language erosion*, a real-life example of language/tool co-evolution.

One might argue that this time is over, that modern languages and tools are much more stable and well engineered. This is unfortunately not true. In the last years the boom in internet-based technologies gives rise to the apparition of a very large number of languages such as scripting languages with internet-based features. These languages are more than ever linked with tool evolution such as web servers. Evolution is rapid and chaotic.

Languages are ill-defined and unstable. The future could be soon populated by legacy web applications raising serious language/program co-evolution issues.

Modelling languages also evolve. This includes in particular the continuous evolution the UML standard over the years. Just like other languages, UML greatly evolves (e.g. UML 1.0 to 1.5 and now 2.0) and presents symptoms of language extension and language contraction. Quoting Warmer about UML 2.0: "*The evolution of UML is absolutely required to make sure that UML will stay up to date with the latest developments in the software industry. The direction taken is guided by the user community, but it requires a big effort* "[26]. This evolution is accompanied by strong co-evolution issues, not only with respect to the large amount of UML diagrams that, but also with respect to the production of large amount of commercial CASE tools. In practice these tools are permanently out of sync. They often deviate from the standard and subtle or most often in important ways.

Component-based development is also getting very popular and component technologies such as COM, .NET or EJB are largely used. These technologies are based on "component models" that defines new concepts and rules that must be followed when developing component-based programs. Though no specific syntax is provided component models could be seen as virtual languages [25]. These languages are often ill-defined, unstable and they greatly evolve because the notion of component is in constant evolution. Once again, this leads to language/ program co-evolution issues. This point is illustrated in Section 7 using Dassault Systèmes as a case study.

Co-evolution is a general phenomenon. It can be applied to requirements, modelling languages, software architecture, etc. The term program is therefore inadequate. Since languages do not even need to be explicit, the same apply to the term language. We use instead more general concepts: models, meta-models and metaware. Roughly put programming language are special cases of meta-models, programs are special cases of models, and programming tools are special case of metaware tools. These concepts and their relationships are described in the remainder of this paper in a systematic way using the 3D software space.

3. The 3D software space

The complex nature of software can be represented as a 3D software product space as depicted in Figure 2. Figures 6, 8 and 9 on the next pages zoom on this space and illustrate its content by means of simple examples. The reader is invited to browse these figures paper to get an overall idea of the content of this space.

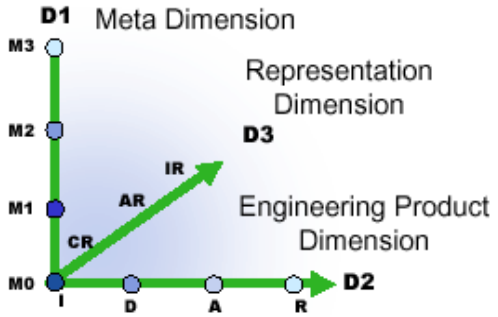
Each dimension corresponds to a different kind of abstraction. All dimensions are orthogonal as it will be show in the next sections.

D1: The meta-dimension. This dimension constitutes the core of the MDA standard. Four levels are distinguished: instances, models, meta-models and meta-meta models. Programs are at the model level (M1), programming-language at the meta-level (M2). The instance level (M0) and the meta-meta level (M3) are included for the sake of completeness. Meta-model/model co-evolution is linked to this dimension.

D2: The engineering product dimension. This dimension aims to structure the software according to each phase in the software life-cycle. It helps for instance to make the distinction between requirement descriptions, architectural documents, and implementation artefacts. Architecture/implementation co-evolution phenomenum is linked to this dimension.

D3: The representation dimension. There are many different ways to represent a given entity ranging from very abstract representations to concrete ones. For instance a programmer might have a mental image of a software architecture. The architecture might also be represented as a boxes-and-arrows graph or as an graph stored in an XML file. Concrete representations heavily depends on the tools that manipulate it. It will be shown that language/tool co-evolution is linked to this dimension.

Each point of this space is represented in the subsequent figures by a cell because it corresponds to a class of software artefacts. Note that the name of each class is conveniently formed by appending the corresponding coordinates in reverse order D3-D2-D1. For instance CR-D-M1 reads "*C*oncrete *R*epresentation of *D*esign *M*odels" and AR-A-M2 stands for *A*bstract *R*epresentation of *A*rchitectural *M*eta-*M*odels.



D1	Meta
M3	Meta-Meta-Model
M2	Meta-Model
M1	Model
M0	Instance

D2	Engineering
R	Requirements
A	Architecture
D	Design
I	Implementation

D3	Representation
IR	Implicit repr.
AR	Abstract repr.
CR	Concrete repr.

Figure 2. The 3D Software Space

In fact, the density of the space is far from uniform. Almost all software artefacts are stuck near to the origin, where programs are. Industry is still code-centric. To illustrate this phenomenon, gray scales are used in most figures. Moreover each dimension will be described as a *pyramid* in the next sections (see Figures 3, 5 and 7). Since the 3 dimensions correspond to a different kind of abstraction, the pyramid structure is well suited to model reality. The *width* of the pyramids represents alternatives or variants, while the *depth* represents the many software entities that constitute each alternative.

4. The meta dimension (D1)

The meta-dimension is surely the most difficult dimension to grasp but it is also the most powerful. It constitutes the core of this paper. In this paper the MDA standard is taken as a reference.

4.1. The meta-pyramid (D1)

The *meta-pyramid* is depicted in Figure 3. A few examples are provided for each level. More examples can be found in Figure 5 in which the meta dimension is represented horizontally.

The most obvious level within the meta pyramid is the model level (**M1**), so let us start by this level. This is the level where regular programs are. This level corresponds to what could be called *appliware*. Entities at this level depends on the particular application domain considered (e.g. banking, nuclear plant design, etc.). For instance the concepts of "account" and "client" might be a part of the banking model, while the concept of "reactor" might be part of the nuclear plant model.

The model level is used to manage the set of all possible real-world situations which are represented at the instance level (**M0**). For instance "Tom" might be a client that owns two accounts "a4099" and "a2394" with a respective balance of $800 and $2000. A point at the instance level describes a particular state of a software at a particular point in time. It corresponds to a program state. Program execution indeed corresponds to the evolution of this state.

Metaware by contrast is independent from application domains. The meta-model level (**M2**) is used to manage the production of software applications. It should describes therefore all software engineering concepts such as "classes", "methods", but also "modules", "frameworks", "configuration", "dynamic libraries", etc. In simple words meta-models capture the set of the concepts used to develop software.

On the top of the pyramid, the meta-meta-model level (**M3**) describes how the meta-models should be described and managed. For instance the MDA standard proposes to use the Meta Object Facilities (MOF) [31]. Simply put, the MOF is a self descriptive subset of UML that allows to describes arbitrary software meta-models (not only the UML meta-model). The MOF is to meta-models what the BNF is to grammars, a standardized way to represent them. Though the meta-meta level is important, this paper concentrates on the meta level for the sake of simplicity. Similarly the term metaware will be used for both level M2 and M3, to avoid introducing the term metametaware.

4.2. Software = Appliware + Metaware

The meta pyramid depicts the realm of software. The next sections will help in making this dimension more concrete, but what is important to understand at this point is that at each level M1, M2, M3 there is some piece of software. Software at the level n+1 is used to build and control software at the level n. Metaware is application-independent software that help producing software applications, that is appliware. A compiler is an example of metaware tools. It is based on the meta-model of the source programming language (e.g. the java meta-model for the javac compiler).

We found distinction between metaware and appliware very important to understand industrial practices. For

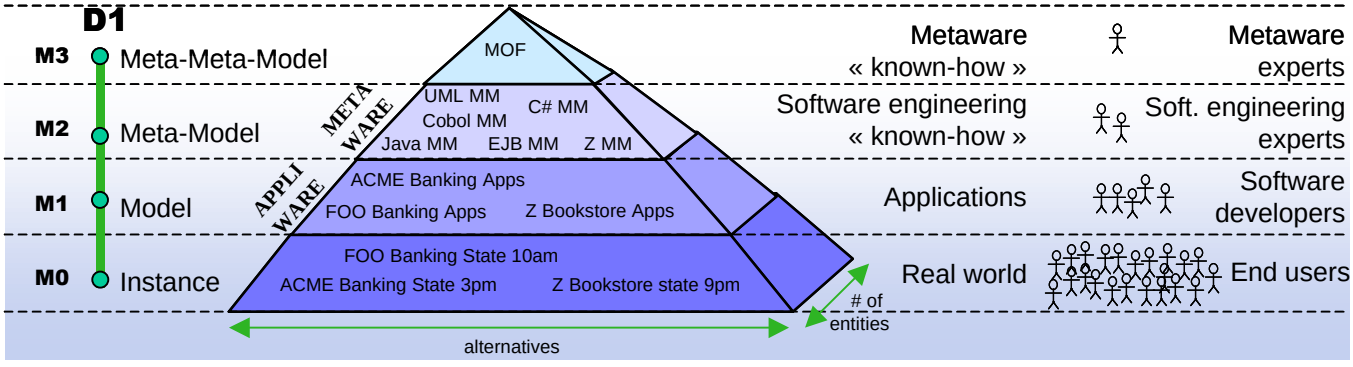


Figure 3. D1: the meta pyramid

Figure 4. D1: the meta-actor pyramid

instance, during our 7-years collaboration with Dassault Systèmes we always stayed at the meta-level. We know much about DS' metaware. By contrast, we never saw DS'appliware [20]. In fact, we never saw a single line of application code in 7 years. Metaware and appliware are distinct parts of software. *Software is metaware plus appliware*. Metaware is software that manage and control software. As it will be shown in the next sections *meta-models are just the visible part of metaware*. Appliware is software that represents applications. Software covers the three higher levels of the pyramid (M1,M2,M3). Level M1 corresponds to appliware, the world of applications. Level M2 and M3 corresponds to metaware. Note that the lower level M0 is about particular states of software execution, which is usually not considered as software.

4.3. The meta/actor pyramid

In fact, one good way to grasp the distinction between the various levels in the meta pyramid is to consider the actors involved at each level. This leads to the *meta/actor pyramid* depicted in Figure 4. The goal of the actors working at the level n+1 is to help actors at the level n to do their job by providing them software. As shown below, raising from a level to the next one decreases the number of people concerned by various orders of magnitude.

End-users are the instance level actors (**M0**). They interact with software applications. They *use* appliware to perform their job. Billions of people around the planet are direct or indirect users of software applications. About 500 000 people use CATIA applications to do their jobs.

Appliware developers are developers of software applications (**M1**). They *produce* appliware for the benefit of M0 actors. They *use* metaware tools to do their job. The number of developers is estimated to be about 6 millions [16]. The great majority of them work at the M1 level. Within the context of DS, more than 1200 software engineers work on developing CATIA applications.

Metaware developers are the meta level actors (**M2**). They *produce* metaware for the benefit of the M1 actors. In practice, each large company includes a separated group of people that define processes, work on quality and build/ integrate tools to manage applications development. They are referred as "know-how providers" in [16]. Their number is estimated to be around 100 000 for the globe [16]. In the context of Dassault Systèmes, these tasks are handled by the Tool Support Team (TST) [20]. This team, made of a few dozens of people, work on metaware and build in-house tools to support CATIA development.

At the higher level of the pyramid, the number of people dealing with meta-meta models (**M3**) is obviously even lower. It might be something around a few thousands for the whole planet because most of the time the meta-meta level is not expressed. However, this could change in the future in particular if the MDA and MOF find their place in industry.

5. The engineering dimension (D2)

Though it is an over-simplification, the waterfall lifecycle clearly shows that software products are not only made of programs: software also includes requirement specifications, global design, detailed design, etc. This leads to dimension D2.

5.1. The engineering pyramid (D2)

Dimension D2 aims at structuring software artefacts following the a very basic engineering process. For the sake of simplicity, only 4 levels are distinguished in the context of this paper, namely the requirement level (**R**), the architectural level (**A**), the design level (**D**), and the implementation level (**I**). This view is obviously a huge simplification of the software realm. The purpose is just to cross this dimension with the other ones, so the model must be simple enough to get understandable results. The next figure shows the *engineering product pyramid*. Each level is illustrated by different examples. More detailed examples are provided in next sections.

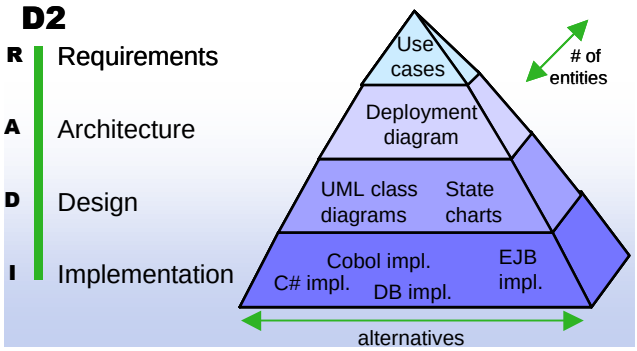


Figure 5. D2: the engineering pyramid

5.2. The engineering / actor pyramid (D2)

Software lifecycles like the waterfall model not only help in identifying the variety of software artefacts. They also make it clear that various actors with different skills are involved in the production of software. Though the engineering/actor pyramid is not depicted, each level of the D2 pyramid involves different actors: requirement engineers, software architects, designers, and developers. A typical project is formed by many developers, but only few architects.

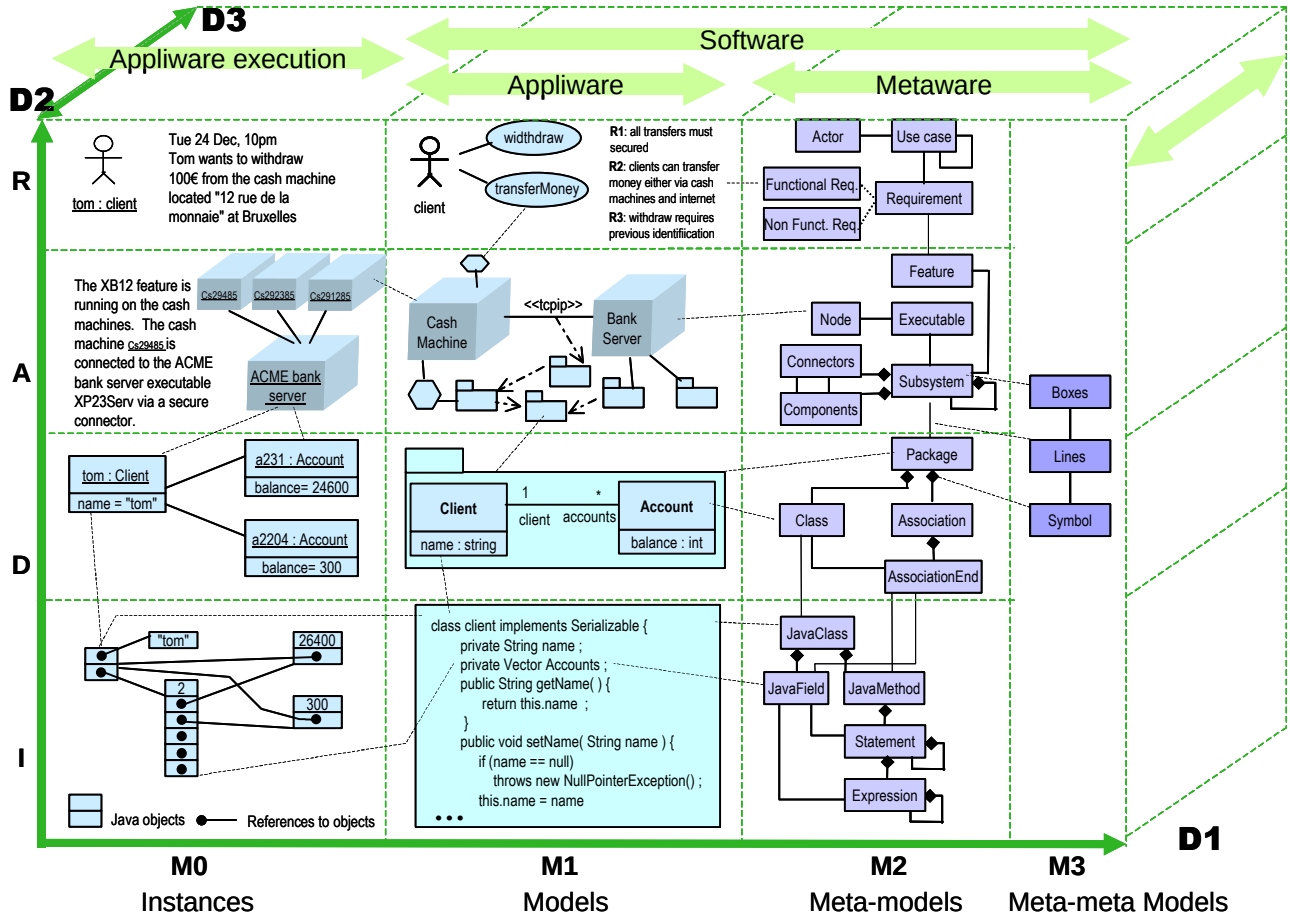


Figure 6. Crossing the meta-dimension (D1) with the engineering dimension (D2)

5.3. Crossing D1 and D2

Confusing the meta dimension and the engineering dimension is quite common, especially when considering the higher levels of abstractions. These two dimensions are however truly orthogonal. For instance, there are architectural models (A-M1), architectural meta-models (A-M2), design model (D-M1), design meta-models (D-M2) and so on. Figure 6 illustrates this property by means of a very simplified yet consistent example. The banking application of the virtual ACME company is considered.

In fact, Figure 6 is centred around column M1 (models) because the engineering process we speak about is defined on models. The reading of the figure should therefore start from that column: other columns are derived from M1. Column M2 acts as the key for the concepts instantiated in column M1. That is, column M2 describes the various meta-models in an informal way, using a UML-like class diagram notation. In fact, boxes, lines and symbols are used to described the meta-models (see column M3). On the opposite side column I describes a particular state of the real world as modelled for the purpose of the ACME banking software. The reader is invited to carefully read Figure 6 which is expected to provided enough intuitive material to grasp the idea.

5.4. Cross-links between levels and co-evolution

All the concepts presented in Figure 6 are connected. However, for the sake of readability only a few links have been drawn between the different cells. The nature of the cross-links depends on the dimension considered.

Vertical cross-links correspond to tracability links between the artefacts produced during the software life-cycle. At the level of meta-model tracability links can just be modelled as regular associations. We first applied this approach to link software architecture and source code in the context of java beans [18], and then in the context of CATIA [22]. Maintaining these links is fundamental to support co-evolution along the engineering dimension (D2), and in particular architecture/implementation(A/I) co-evolution. Tracability between models is considered as an important issue in the MDA approach [14].

Horizontal cross links are different in nature: they relate an entity to its model and conversely a model to its instances. The modelling of these cross-links constitutes the basis to support co-evolution along the meta-dimension and in particular meta-model/model (M2/M1) co-evolution.

6. The representation dimension (D3)

The reader might have noticed that all the examples in Figure 6, do not correspond to the same kind of representations. In fact one can imagine many other alternative representations for each cell. What is needed is an additional dimension to represent these variations. This leads to D3, the "representation dimension" (D3).

6.1. The representation pyramid (D3)

It is important to recognize that a single piece of information can be represented in many different ways ranging from implicit representations to very concrete ones. The fact that a piece of data is not explicitly represented as a sequence of bits does not mean that it does not exist. For instance, most of the time, software architecture is not explicitly represented. Software architects maintains some mental images and this might be enough. To communicate, box-and-arrows diagrams are often used. Other information is also represented by means of natural languages. Or it is simply part of the "implicit knowledge" of a given company. These kind of representations are obviously not adapted to automated processing. Very concrete representations are required when tools support is needed.

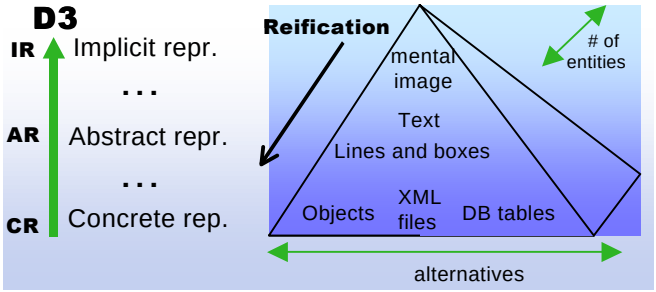


Figure 7. D3: the representation pyramid

The figure above depicts the *representation pyramid*. Though there is a continuum of abstraction levels, only three levels are named for the sake of simplicity: implicit representation (**IR**), abstract representation (**AR**) and concrete representation (**CR**). The lowest level is oriented towards tool processing, while the highest level represents implicit knowledge. Though the actor pyramid is not depicted human actors would be at the top of the pyramid while the many tools that process concrete representations would be at the lower level.

The shape of the pyramid is justified by the fact that a very large set of representation techniques can be used to represent a particular software entity. This includes for instance graph of objects in memory, tuples in a database, XML files, etc. Concrete representations greatly vary depending on the purpose of the tool considered. For instance a compiler, a syntax editor and a test coverage tool might represent the same program with very different internal structures. That's just a matter of concrete representation.

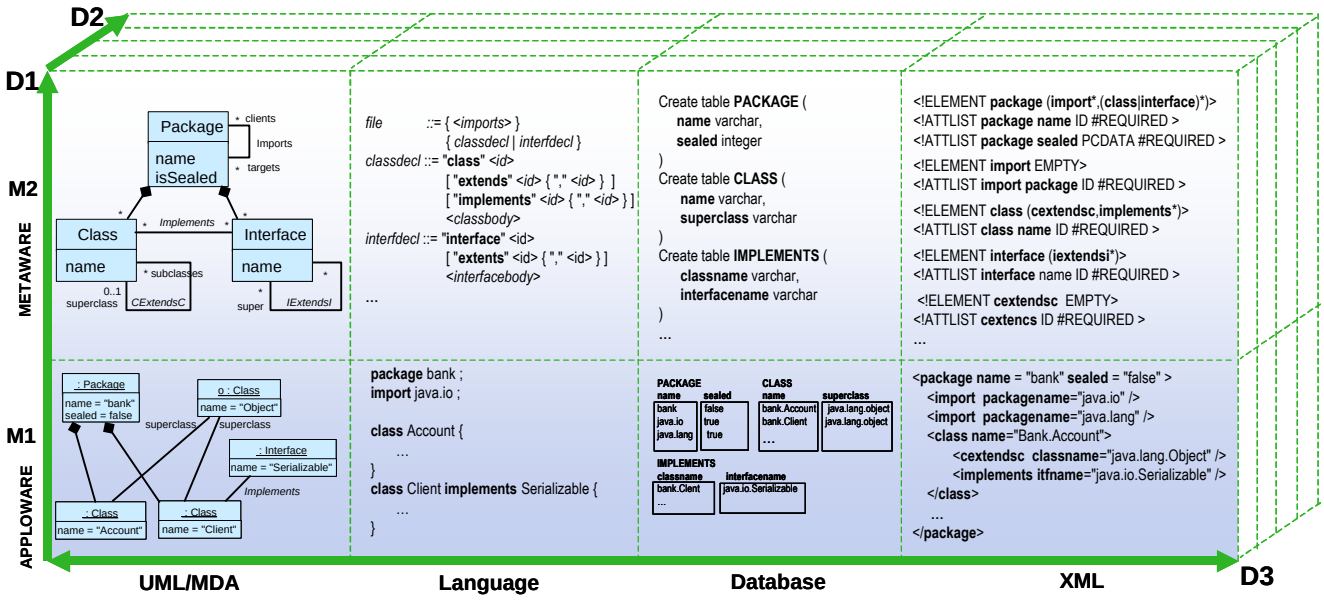


Figure 8. D1+D3: Alternatives representations of a Java program (I-M) and a Java meta-model (I-MM)

6.2. Crossing D3 and D1-D2

Though this might not be obvious, the representation dimension D3 is orthogonal both to the engineering dimension D2 and the meta dimension D1. Due to space constraint only small slices of the space could be provided.

Figure 8 on the previous page illustrates the variety of concrete representations both for a java program (M1) and a java meta-model (M2). At both levels, the *same* information is represented in different ways. Since all alternative representations are more or less at the same level of abstraction with respect to D3, this example illustrates the width of the representation pyramid but not its height.

The continuum from implicit representations to very concrete ones is illustrated in Figure 9 for the metaware column (M2). This small slice of the software space illustrates in particular the notion of *conceptual meta-model*, *specification meta-model* and *implementation meta-model* as well as *metaware tool*. Due to limitation space, Figure 9 illustrates the height but not the width of D3 pyramid: only one possible representation is selected when going down from one level to the next one.

A very simple example of meta-model (a small subset of the java language) has been selected for the sake of clarity. In practice the approach has to be applied on much more complex meta-models, such as proprietary architectural meta-model (e.g. [20][27])[1].

The implicit knowledge a java programmer could have about the java language would fit on the top of the pyramid. A programmer might know for instance that java provides simple inheritance between “classes”, yet a “class” may implement multiple “interfaces”. This is the implicit part of the metaware. Just implicit knowledge.

At the other extremity of the spectrum, we found very concrete metaware artefacts managed by metaware tools. In the case of a programming language, metaware tools include all tools that parse, analyse, interpret, and manipulate programs: interpreters, compilers, browsers, etc. Obviously, each tool have an embedded knowledge of the language it manipulates. This knowledge is represented somehow in the code of the tool. This observation is consistent with what Lammel and Verhoef report in [24].

As show in Figure 9, the role of meta-models is central to metaware. *Meta-models makes the bridge between concrete metaware items and informal metaware knowledge*. Meta-models constitute the conceptual part of the metaware. In the academic world, the term meta-level usually evokes this part, because meta-models are neat abstractions to reason about. Unfortunately our experience

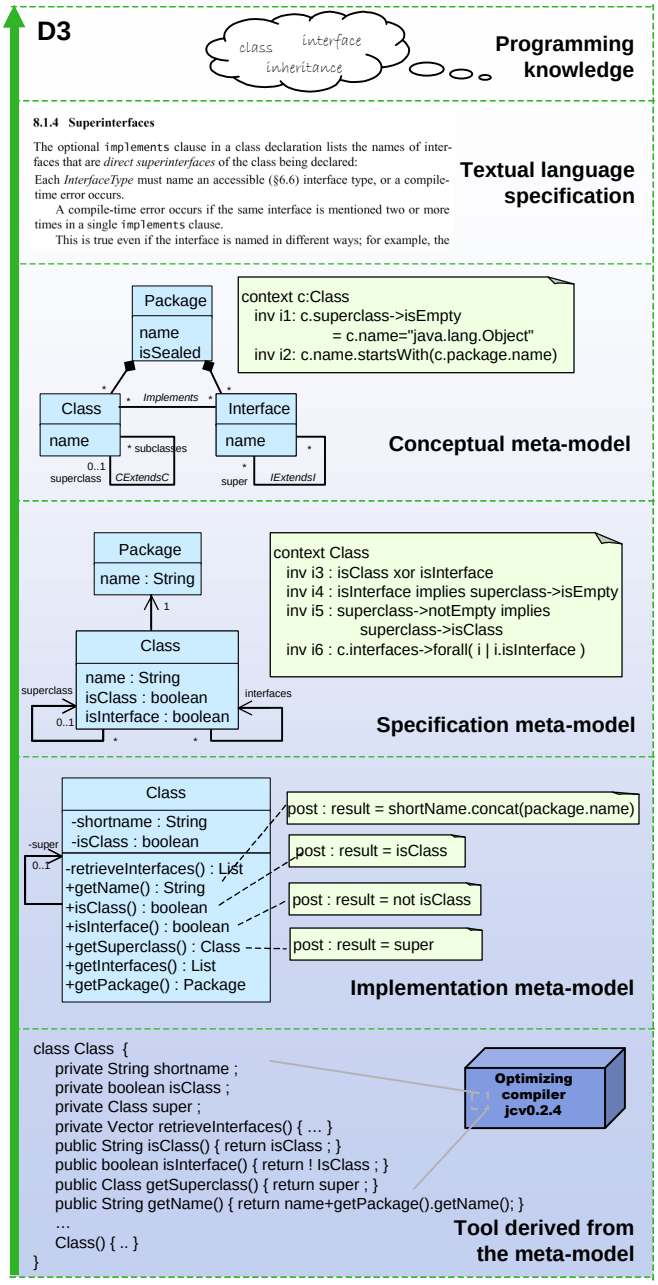


Figure 9. D3: Metaware

As show in Figure 9, the role of meta-models is central to metaware. *Meta-models makes the bridge between concrete metaware items and informal metaware knowledge*. Meta-models constitute the conceptual part of the metaware. In the academic world, the term meta-level usually evokes this part, because meta-models are neat abstractions to reason about. Unfortunately our experience shows that this part is often missing in industry. Though the term meta-level might not evoke anything in large companies, metaware *does* exist. It takes however the form of software development tools. Many of these tools are complex and often proprietary [20]. Recovering meta-models is important in particular since meta-models capture the application independent part of the company know-how [32].

Finally it should be noted that the distinction made between conceptual meta-model, specification meta-model and implementation meta-model is indeed based on the

1. For a full understanding of the various steps described in Figure 9 it is assumed that the reader has both a knowledge of java and the understanding of the refinement process as described in [30]. .

application of the principles introduced by Fowler [30]. These levels are usually applied on models to develop appliware through successive refinement. We found however these concepts very useful to categorize existing meta-models. In fact, reading Figure 9 from bottom to top clearly suggests a forward engineering process, while reading the figure from top to bottom leads to a reverse engineering process. While D2 is centred around appliware engineering, D3 is centred around *metaware engineering*.

7. Evolution: entering the fourth dimension

As depicted in Figure 10, evolution can be introduced in the conceptual framework by adding a fourth orthogonal dimension representing time.

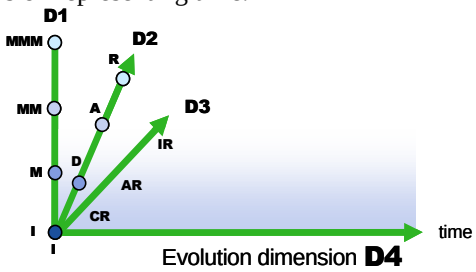


Figure 10. Entering the fourth dimension

This modelling put emphasis on the fact that every classes of software artefacts evolve. Everything evolve or will evolve soon or later. Large companies with a long background about software development know that. Along the years and decades they accumulate know-how about evolution-in-the-large. However, stability is still a very common yet implicit assumption made in many research projects. We are not aware for instance of much research work concerning meta-model evolution.

Co-evolution phenomena can easily represented by crossing one abstraction dimension with the time dimension. A pair of cells X and Y leads to co-evolution that will be noted X/Y-CoE. Figure 11 depicts how evolution interact with the engineering dimension revealing for example architecture/implementation co-evolution (i.e. A/I-CoE). The figure also suggests that the rate of changes greatly vary between software artefacts. For instance, the architecture of a software is expected to be much more stable that its implementation.

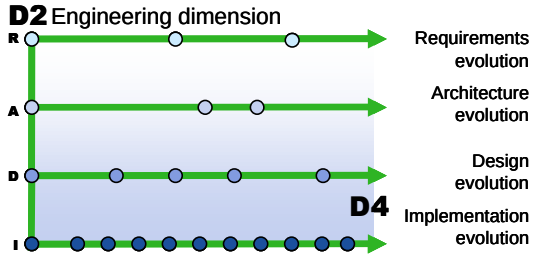


Figure 11. Crossing D2 and D4

Figure 12 add time to the meta-dimension. Notice that instance evolution (M0-E) corresponds to program execution. The rate of change is therefore extremely high, especially when compared with higher level of abstractions. Similarly models (e.g. programs) are much more unstable than meta-models (e.g. languages).

The conceptual framework is useful to structure ideas but it is sometimes too abstract to get a real feeling of what happen in practice. Let us illustrate the concept of meta-model and model co-evolution (M2/M1-CoE).

8. Example of M2/M1 co-evolution in industry

From our collaboration with Dassault Systèmes we can draw various conclusions about large scale software development and evolution. Most conclusions could also apply to other industrial contexts as well.

(1) Evolution-in-the-large is often achieved through ad-hoc processes, tools and concepts. This should not be surprising because many problems are discovered on the run. Pragmatic solutions are incrementally elaborated in a "as-needed" mode and sometime in "panic" mode to solve unexpected issues. For instance, in [20] we describe how ADELE, the configuration management tool developed by our team, was adopted at large by Dassault Systèmes and how the huge requirements in collaborative development lead to "hot" periods. Large companies where thousands of developers work on the same software cannot stop their development process when they find problems. They have to find solutions.

(2) Architecture is fundamental to evolution-in-the-large but its explicit representation with ADLs raises more problems than it solves. Industry is code centric and most architectural facts should be extracted from the code. Initially one of our goals was to study what kind of ADLs could be applied to support the evolution of CATIA [23][28]. We soon discover however that a much better approach was to provide architectural recovery and software exploration tools [22][29].

(3) The notion of software architecture is really more complex than the academic vision tend to explain and its exact nature really depends on the company culture and know-how [23]. In particular, we didn't found a single definition of architecture really helpful in practice. We

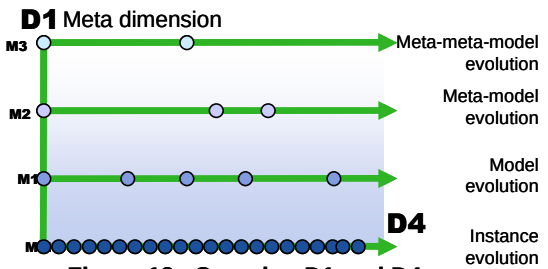


Figure 12. Crossing D1 and D4

found on the contrary that many of the architectural concepts used at-large within DS were beyond traditional concepts. For instance we identified the business architecture. It describes how software can be sold in parts. This structure is quite complex at DS.

(4) The company "know-how" is in part immaterial knowledge shared among the company, but in part materialized by in-house metaware tools. These tools support appliware development and constrain appliware evolution by enforcing specific processes and in-house quality standards. These tools are either bought and then customized, or developed internally by the tool support team. Over the last decades DS has developed a huge amount of metaware to support for instance configuration management, testing, component-based-development, etc. This range from sophisticated tools to hand-craft tools .

(5) The distinction must be made between metaware and appliware, between M2 and M1, between models and meta-models, especially in the context of software architecture. Too often these levels are confused in this context; in large part because architecture if a fuzzy notion. The distinction must be made between architectural models (A-M1), which are application dependent, and architectural meta-models which represent reusable know-how about building software (A-M2). This difference is illustrated in Figure 6 in the respective cells. Hofmeister and her colleagues describe reusable architectural know-how resulting from Siemens experience in [20]. This book is organized around 4 meta-models presented on the front and back covers of the book. Extracts of the architectural meta-models we recovered from DS' metaware can be found in [22][23].

(6) Everything evolve in large companies. In particular the notion of architecture and the architecture of applications. To be more precise both architectural meta-model evolution (A-M2-E) and architectural model evolution (A-M1-E) take place. In the first case this is the architectural know-how which evolves, in the second case this is the architecture of particular applications. This leads to architectural meta-model/model co-evolution issues (A-M2/M1-E). That is, without entering into the implementation details (I), one can observe that both architectural concepts and their occurences in software applications evolve.

(7) The evolution of the architectural concepts can have a strong impact on the architecture of the application, but also the other way around. For instance in the mid 90's DS decided to develop a component technology similar to Microsoft' COM but with also a set of unique features to cope with DS specific needs [21]. This technology evolved at the same time as the applications built using it. DS know-how about component-based architectures greatly evolved with the experience gained in developing large set of components (about 8000 today). Note that when the concepts underlying the component technology change, component-based applications may or may not be impacted. For instance sometimes an architectural concept reveals to be harmeful after a long period of use without noticable problem. This was the case for instance for a feature included in the DS component technology. This feature greatly simplified component developement but it later revealed to be responsible of significant decreases in performance when used at-large. DS then decided to remove it from the set of features available to develop software. From a conceptual point of view this corresponds to a removal of an element from the architectural meta-model. Components using this feature had to be identified to be upgraded. Sometimes, external events make it necessary to improve the architectural meta-model. For instance a few years ago DS decided to make its component technology available to partners such as Boeing. Before this DS used a visibility model based on the traditional public/ private distinction to control dependencies between software entities. This was enough within the context of DS, but not enough for externalisation because more levels had to be added to better control external dependencies. From a conceptual view, this modification just imply at the level of meta-model to change the type of the "visibility" meta attribute, as well as to update the constraints associated with this attribute in the meta-model. From a concrete point of view, the metaware tool that control dependency management was modified and the level of visibilty had to be assigned for each software entity concerned. From a conceptual point of view, this modification consiste in updating in architectural models the value of the visibility attribute of each entity concerned.

(10) *Metaware tools developed within large companies are often built in an incremental and in ad-hoc way, following the needs of the company*. These tools are often hand-craft using for example unix scripts to automate transformations. As shown in the previous example some transformations occur only a few time and building a tool from scratch could be too costly. One important issue in this context is to facilitate the production of metaware. Declarative meta-progamming or using meta-model driven environment are very promizing approaches in this context.

9. Conclusion

Software evolution is too often confused with program evolution. Software is much more than programs. Just like programs, languages follow Lehman's laws of continuing changes: in order for a *language* to continue to be useful (and used) in the real world it must change continuously. Languages are integral part of software. Languages, tools and programs evolve in parallel.

While architecture and implementation co-evolution has

been identified as a natural process during evolution-in-the-large, this paper has unveiled the existence of meta-model and model co-evolution, which is a generalisation of language/program co-evolution. We shortly described for instance the architectural meta-model/architectural model co-evolution problem as it occurs in industry.

These complex issues has been studied thanks to the provision of conceptual framework. The framework is based on the fact that software artefacts can be classified along a three abstraction dimensions. The meta-dimension is based on the four layers and includes models and meta-models. The engineering dimension distinguishes software artefacts according to the phase in which they are produced. The representation dimension makes it possible to model artefacts that range from implicit and fuzzy knowledge to very concrete representations used by tools.

Emphasis has been put on the need to make the distinction between metaware and appliware. Appliware is the set of applications, while metaware is software that help in developing and controlling appliware. In fact, software is metaware plus appliware. Software evolution should not be restricted to appliware evolution, metaware also evolves.

Some experts predict that the MDA standard could have a strong influence on the future of software engineering [17][14]. However, failure is still possible. Historically, most approaches looking only towards the future have failed. Roughly put, while ADLs were designed to make the architecture explicit (and failed), Model Driven Engineering is designed to make explicit models and meta-models. This is certainly the way to go, but this raises some questions. Will software engineers accept to draw UML models if evolution is not supported in a very effective way? What about extracting models and meta-models from existing software? What about model and meta-model co-evolution in the context of the MDE? Could we assume that "standard" meta-model will not evolve in the long run? What about reverse engineering of meta-models from legacy and proprietary metaware?

We see Model Driven Engineering as a very promising approach. But we also believe that this approach will fail if evolution is poorly supported and if legacy software is not taken into account. Metaware evolution and metaware reverse engineering are open research issues as well as effective tool support for meta-model/model co-evolution.

10. References

[1] T. Mens, J. Buckley, M. Zenger, A. Rashid, "Towards a Taxonomy of Software Evolution", USE 2003

[2] M. Felici, "Taxonomy of Evolution and Dependability", Workshop on Unanticipated Software Evolution, USE'2003.

[3] L. O'Brien, C. Stoermer, C. Verhoef, "Software Architecture Reconstruction: Practice Needs and Current Approaches", SEI Technical Report CMU/SEI-2002-TR-024, 2002

[4] A.E. Hassan, R.C. Holt, "Architecture Recovery of Web Applications", ICSE 2002

[5] S. Boucetta, H. Hadjami, F. Kamoun, "Architectural Recovery and Evolution of Large Legacy Systems", IWPSE 1999

[6] Q. Tu, M.W. Godfrey, "An Integrated Approach for Studying Architectural Evolution", IWPC 2002

[7] J.B. Tran, M.W. Godfrey, E.H.S. Lee, R.C. Holt, "Architectural Repair of Open Source Software", IWPC 2002

[8] J. Zhao, H. Yang, L. Xiang, B. Xu, "Change impact analysis to support architectural evolution", Journal of Software Maintenance and Evolution, 14:317–333, 2002

[9] R. Wuyts, "A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation", PhD, Vrije yiversity of Brussel, 2001.

[10] K. Mens, T. Mens, M. Wermelinger, "Supporting unanticipated software evolution through intentional software views", USE 2002

[11] T. D'Hondt, K. De Volder, K. Mens, R. Wuyts, "Co-evolution of Object-Oriented Software Design and Implementation", Proc. Int'l Symp. Software Architectures and Component Technology: The State of the Art in Research and Practice, Kluwer, 2000

[12] OMG, "MDA: the OMG Model Driven Architecture", http://www.omg.org/mda/

[13] OMG, "Model Driven Architecture - A Technical Perspective", ormsc/01-07-01, 2001

[14] A. Kepple, J. Warmer, W. Bast, "MDA Explained - The Model Driven Architecture: Practice and Promise", Addison Wesley, 2003

[15] S. Kent, "Model Driven Engineering", LNCS 2335, 2002

[16] X. Blanc, P. Desfray, "Model Driven Engineering", in french, to appear in 2003

[17] J. Bézivin, X. Blanc, "MDA: Towards an Important Paradigm Change in Software Engineering", in french, Développeur Référence, http://www.devreference.net/Develop, July 2002

[18] V. Marangozova, "Linking the Software Architecture with Source Code", Master, in french, University of Grenoble, June 1998

[19] Softeam, "Guarantee permanent Model/Code consistency: Model driven Engineering versus "Roundtrip engineering", 2000

[20] J.M. Favre, J. Estublier, R. Sanlaville, "Tool Adoption Issues in Very Large Software Company", 3rd Workshop on Adoption Centric Software Engineering, ACSE 2003

[21] J. Estublier, J.M. Favre, R. Sanlaville, "An Industrial Experience with Dassault Systèmes' Component Model", Book chapter in Builiding Reliable Component-Based Systems, I. Crnkovic, M. Larsson editors, Archtech House publishers, 2002

[22] J.M.Favre *and al.*, "Reverse Engineering a Large Component-based Software Product", CSMR'2001

[23] R. Sanlaville, "Software Architecture: An Industrial Case Study within Dassault Systèmes", PhD dissertation, in french, Univeristy of Grenoble, 2002

[24] R. Lammel, C. Verhoef, "Semi-automatic grammar recovery", Software Practice and Experience, 2001

[25] J. Estublier, J.M. Favre, "Component Models and Component Technology", Book chapter in Builiding Reliable Component-Based Systems, Archtech House publishers, 2002

[26] J. Warmer, "The Future of UML", available from www.klasse.nl

[27] C. Hofmeister, R. Nord and D. Soni. Applied Software Architecture. Addison-Wesley Publisher, 2000.

[28] Y. Ledru, R. Sanlaville, J Estublier, "Defining an Architecture Description Language for Dassault Systèmes", 4th International Software Architecture Workshop, 2000.

[29] J.M. Favre, "GSEE: a Generic Software Exploration Environment", 9th International Workshop on Program Comprehension, IWPC'2001

[30] M. Fowler, "UML distilled: A brief guide to the standard modelling language", Addison Wesley, 1999

[31] OMG, "Meta Object Facilities (MOF) Specification, Version 1.4", April 2002

[32] P. Desfray, "MDA – When a major software industry trend meets our toolset, implemented since 1994", Softeam white paper, 2001