Model Driven Software Development – State of The Art and Perspectives

(Invited paper)

Branko Perisic
Computers and Control Department
The University of Novi Sad, Faculty of Technical Sciences,
Novi Sad, Serbia
perisic@uns.ac.rs

Abstract— **Modern day system developers have some serious problems to cope with. The systems they develop are becoming increasingly complex as customers demand richer functionality delivered in ever shorter timescales. If we simply replace word *system* with *software* we arrive to the origin of eternal software development questions and dilemmas. The evolution of software engineering discipline is characterized by the methodology proposals aiming to answer these questions mainly sublimated as *how to produce working software artifacts according to the customer needs, in time and within the budget*? According to that The Rapid Software Development Methodologies became a challenging issue for Software Engineering discipline. In this paper Model Driven Software Development is analyzed from two underlining aspects: *the state of the art* and *the future perspectives*.**

Keywords-Software; Metamodeling; Model Driven Software Development; Model Driven Architecture; Software Development Methodologies;

I. INTRODUCTION

.

If we look to the history of software development it can be said that its professional age starts with the formulation of Software Engineering Body of Knowledge (SWEBOK).

In 1958, John Wilder Tukey, one of the most influential statisticians, has introduced the term *software.* ISO/IEC/IEEE Systems and Software Engineering Vocabulary (SEVOCAB) defines *software engineering* as "*the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software*".

Contemporary system developers have some serious problems to cope with. The systems they develop are becoming increasingly complex as customers demand richer functionality delivered in ever shorter timescales.[2] They are exposed to a huge diversity of implementation technologies, design techniques and development processes, particularly the latest *'silver bullet'* design abstraction - Model Driven Software Development.

Looking from Tukeys' perspective, during the past 56 years, software engineering has evolved from a conference catchphrase (*the term software engineering was used in the title of a NATO conference held in Germany in 1968.*) into an matured engineering profession, characterized by [1]:

- **a professional society** (The IEEE Computer Society first published its Transactions on Software Engineering in 1972);
- **standards that specify generally accepted professional practices** (A committee for developing software engineering standards was established within the IEEE Computer Society in 1976. The standard was completed in 1995 with designation ISO/IEC 12207 and given the title of Standard for Software Life Cycle Processes.)
- **Software Engineering Body of Knowledge** (SWEBOK)(The IEEE version of 12207 was published in 1996 and provided a major foundation for the body of knowledge captured in SWEBOK 2004. The current version of 12207 is designated as ISO/IEC 12207:2008 and IEEE 12207-2008 and provides the basis for SWEBOK V3.)
- **a code of ethics**;
- **conference proceedings**;
- **textbooks**;
- **curriculum guidelines and curricula**;
- **accreditation criteria and accredited degree programs**;
- **certification and licensing.**

A. *Software Development as Goal-directed Cooperative Game of Inventoion and Communication*

An alternative underlying model for software development is presented in [3]. According to Cockburn's article, software development may be seen as a series of resource-limited, goal-directed cooperative games of invention and communication. The primary goal of each game is the production and deployment of a software system; the residue of the game is a set of markers to assist the players of the next game. The next game is an alteration of the system or the creation of a neighboring system. Each game therefore has as a secondary goal to create an advantageous position for the next game.

Since each game is resource-limited, the primary and secondary goals compete for resources. The cooperative game model provides the benefits that the software engineering model misses:

- It raises to the proper priority level issues crucial to successful software projects;
- it explains how teams with messy-looking processes sometimes outperform others with tidier processes;
- it helps busy practitioners decide how to respond to unexpected situations.
- it is seen that much of engineering in the general belongs in the category of resource-limited, cooperative games [3].

B. *The Levels of Abstraction and Reusability*

The history of software development is a history of raising the *level of abstraction* and the *level of reusability.*

Raising the level of abstraction changes the platform on which each layer of abstractions depends [4].

Raising the level of reuse assumes that the teams share defined interfaces, working to build components that can simply be plugged together at the end of the project. Components and frameworks are rarely plug-and-play, and teams can spend inordinate amounts of time writing "glue code" to stick components together properly [4].

Components and Frameworks are not at the highest reusability level. More reuse may be found concerning databases and data servers and general services that rely on implementation technologies.

C. *Programming and Modeling Today*

It is essential to distinguish Software Development from Programming. Software development includes aspects such as requirements engineering, development processes, software design, and documentation. The programming is usually seen like an act of constructing creating and testing an instance of implementation.

The main tools for programming today are general purpose programming languages such as Java, C#, C/C++, Piton, Scala or Ruby. They typically use procedural, functional, or object oriented abstractions in various combinations. However, to build nontrivial applications, any number of additional languages and formalisms are required, including XML, HTML or state charts.

The use of Frameworks is in its expansion. In the Java world, prominent examples include JEE, Spring or Hibernate. They typically come with their own configuration "languages", usually XML-based.

Unfortunately, the integration between the various languages, frameworks, and technologies is often limited and usually based on name references.

Architectural concepts cannot be represented as first-class entities in programs, leading to all kinds of maintainability problems as well as limited analyzability and tool support due to the existence of:

- hierarchical components and instances,
- ports,
- protocols,
- pre- and post conditions,
- messages,
- queue,
- data replication specifications,
- persistence mappings, or
- synchronization.

For example Software Components are often represented as: *all the classes in a package*, *a façade class that is registered in some kind of configuration file* or maybe an *XML descriptor* that somehow captures the metadata about the component.

On the other hand, **model** represents a specification of the function, structure and behavior of a system within a given context, and from a specific point of view. It is often represented by a combination of drawings and text, augmented where appropriate with natural language expressions.

Modeling leads to formal specification building based on the language that has well defined meaning associated with each of its constituents. This formalism

distinguishes the model from a simple *drawings* and allows it to be expressed in a well-defined format, such as XML, in accordance with an underlining XMI schema.

II. METHODOLOGICAL ASPECTS OF ENGINEERING SOFTWARE

Engineering disciplines tend to use the terms method and methodology interchangeably. From this perspective a development method may be defined as a codified conceptual framework that aids the act of applying a systematic approach to create a set of interrelated artifacts, which ultimately lead to the system that shall exhibit either stated or unstated properties. The Development Method encapsulates The Development Process and The Development Artifacts.

MDE has several representative research movements and institutionalized paradigms that are unavoidable now or in the future.

They are: The Model Driven Software Development (MDDS), Model Driven Engineering (MDE), Model Integrated Computing (MIC), Language Oriented Programming (LOP), Domain-Specific Modeling (DSM), Generative Programming, Framework Specific Modeling (FSM), Executable UML (xUML), The OMG Model Driven Architecture (MDA), Agile MDA, etc.[6,7,9] Domain analysis, meta modeling, model-driven generation, template languages, domain-driven framework design, and the principles for agile software development form the backbone of this approach, of which OMG's MDA is a specific flavor.

A. MDSD – Model-Driven Software Development

Model-Driven Software Development is a software development approach based on the idea of developing software from domain-specific models. The ultimate aims of MDDS is the improving productivity and maintainability of software by raising the level of abstraction from source code, written in a general purpose language, to high-level, domain-specific models such that developers can concentrate on application logic rather than the inherent complexity of low-level implementation details [6,7,9].

Considering the modeling paradigm, as an approach to the rapid software development, there are two basic directions that constitute ***the state of the art*** and ***the perspectives*** of software development. [6]

From the point of view of software designers there are two main roles assigned to models and modeling. One treats underlining model as an ***analytical tool*** for better understanding of problem domain, requirements analysis, design specification and overall project documenting.

The other treats the model as an ***high level executable*** that constitutes the foundation for automatic code generation on targeted implementation platform.

B. MBE Model- Based Engineering

Model-Based Engineering (MBE) is a key enabling technology for developers that seek the transition from traditional systems development processes, that are *document-based* and *code-centric,* to more effective processes that are r*equirements-driven* and *architecture-centric*.

MBE is an umbrella term that subsumes several sub-disciplines: Model-Driven Development (MDD), which focuses on software-intensive applications; Model-Based Systems Engineering (MBSE), which focuses on Systems Engineering applications; Business Process Modeling (BPM), which focuses on Business Analysis applications; and Ontology Engineering (OE), which focuses on Knowledge Engineering applications. (Figure 1.)

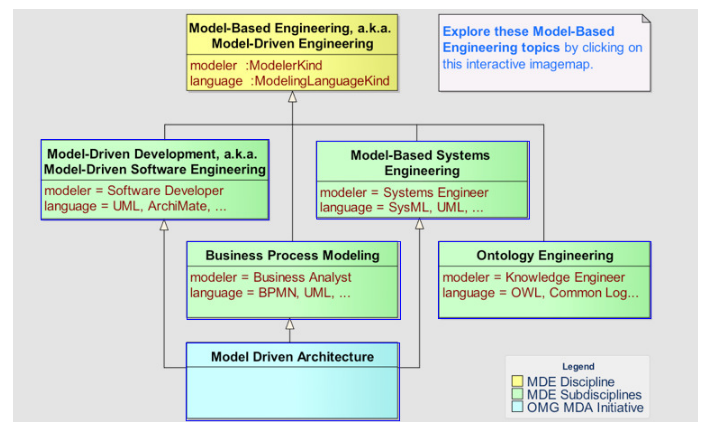

Figure 1. The Object model of models relationships

MBE, MDE and MDD, may be used interchangeably, and are usually treated as software and systems development paradigms that emphasize the application of visual modeling principles and best practices throughout the System (or Software) Development Life Cycle (SDLC).

MBE encourages developers to use models to describe both the problem and its solution at different levels of abstraction, and provides a framework for methodologists to define what model to use at a given level of abstraction, and how to lower the level of

abstraction by defining the relationship between the participating models.

An MBE process should define:

- how many levels of abstraction are there, and what platforms have to be integrated;
- what are the modeling notations and the abstract syntax to be used at each level of abstraction;
- how refinements are performed, and what platform and additional information they integrate into the lower level of abstraction;
- how code is generated for the modeling language used at the lowest level of abstraction, and perhaps even how to deploy that code;
- how can a model be verified against the upper level model, how can it be validated, and how can it generate test cases for the system under development.

C. MIC – Model-Integrated Computimg

Model-Integrated Computing (MIC) - has been developed over two decades at ISIS, Vanderbilt University, for building a wide range of software systems. [36]

MIC focuses on the formal representation, composition, analysis, and manipulation of models during the design process. It refines and facilitates "model-based development" by providing three core elements:

- the technology for the specification and use of domain-specific modeling languages (DSML);
- the fully integrated meta-programmable MIC tool suite; and
- an open integration framework to support formal analysis tools, verification techniques and model transformations in the development process.

Software and systems development in the MIC framework includes three technology components:

- technology for building, analyzing, and managing models;
- technology for transforming models into executable programs and/or analyzable for system engineering tools; and
- technology for integrating applications on heterogeneous parallel/distributed computing platforms.

The MIC Software and System Development process is comprised of three levels:

- The *Application Level* represents the synthesized, adaptable software applications. The executable programs are specified in terms of a Composition Platform (e.g. CORBA, Multigraph Computational Model (MCM) and others).
- The *Model-Integrated Program Synthesis Level (MIPS)* comprises domain specific modeling languages (DSML) and tool chains for model building, model analysis, and application synthesis.
- The *Meta-Level* of MIC provides metamodeling languages, metamodels, metamodeling environments and metagenerators for creating domain specific tool chains on the MIPS level. [36]

According to the generic components of the MIC tool architecture are:

- Generic Model Environment (GME),
- Model Management tool suite (UDM),
- Model Transformation tool suite (GReAT), and
- Design Space Exploration tool suite (DESERT).

D. LOP - Language Oriented Programming

Language oriented programming (LOP) is a style of computer programming in which, rather than solving problems in general-purpose programming languages, the programmer first *creates one or more domain-specific languages for the problem*, and *solves the problem* in those languages.

The concept of language oriented programming takes the approach to capture requirements in the user's terms, and then to try to create an implementation language as isomorphic as possible to the user's descriptions, so that the mapping between requirements and implementation is as direct as possible.

A measure of the closeness of this isomorphism is the "redundancy" of the language, defined as the number of editing operations needed to implement a stand-alone change in requirements.

It is not assumed *a-priori* what is the best language for implementing the new language. Rather, the developer can choose among options created by analysis of the information flows — what information is acquired, what its structure is, when it is acquired, from whom, and what is done with it.

E. DSM – Domain-Specific Modeling

Domain-Specific Modeling raises the level of abstraction beyond programming by specifying the

solution directly using domain concepts. The final products are generated from these high-level specifications. Industrial experiences have consistently shown DSM to be 5-10 times more productive than current software development practices. [10]

According to Software Productivity Research, the average productivity in Java is only 20% better than in BASIC. C++ fares no better than Java. If we go back a couple of decades more, there is a radical change: a leap in productivity of 400% from Assembler to BASIC. The 400% increase was because of a **step up to the next level of abstraction**. Each statement in C++, BASIC or Java corresponds to several Assembler statements and, more importantly, program statements, written in these languages, can be automatically translated into Assembler. [10]

Traditional modeling languages, like UML, have not increased productivity, since ***the core models are on the same level of abstraction as the programming languages supported*. *UML tries to be all things to all men, and thus cannot raise the level of abstraction above the lowest common denominator.*** UML has its benefits because of visual nature that makes diagrams more expressive and easy to read and analyze.

Domain-Specific Modeling raises the level of abstraction and hides today's programming languages, in the same way that today's programming languages hide assembler. This is a whole level of abstraction higher than UML, and makes each symbol worth several lines of code. The application is automatically generated from these high-level specifications with domain-specific code generators, aided where necessary by existing component code.

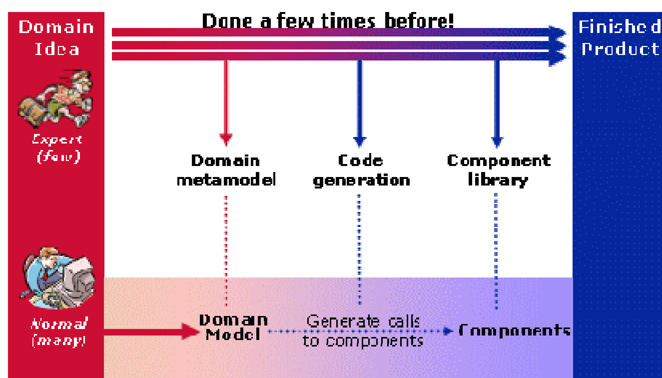

Figure 2. DSM Foundation

As an expert has specified the code generators, they produce products with better quality than could be achieved by normal developers by hand. In order to support DSM with full automatic code generation it is essential to develop three main components (See Figure 2.):

- a domain-specific modeling language and editor;
- a domain-specific code generator; and
- a domain-specific component library.

1) Developing the domain-specific modeling language and editor

In MDE, a Domain-Specific Language (DSL) [8] is a specialized language, which, combined to a transformation function, serves to raise the abstraction level of software and ease software development. The experience and intuition of the expert, combined with hints from the component library, domain rules and architects are the real sources of clues. Metamodeling languages can be applied here to describe both the domain rules and their mappings. A toolset that allows rapid prototyping is practically a necessity because it enables creating a part of the metamodel as a prototype, and instantly test it by making an example model.
However, in practice, beyond this general definition, DSLs adopt multiple forms of representation and implementation. The reason of a DSL feature model is to formalize DSL and DSL tool variants:

- A first application of this feature model is a DSL tool factory, which applies variations during production of DSL tools
- A second application is the selection of pertinent DSL families among all possible families from the feature model. A third application is the definition of DSL tool foundations. A fourth usage is the selection of DSL tools.[8]

2) Developing the code generator

The code generation definition forms the final task, conceptually if not chronologically. In practice there will be a large degree of parallelism and incrementality between all three tasks. The DSM tool should provide the necessary functionality for creating such generation scripts, and should guide the expert where possible by allowing him to reference and use the concepts in the metamodel.
Of all the phases, code generation probably varies the most between domains. In some domains it will be possible to produce a large fraction of code with a relatively simple code generation scripting language, such as is already provided in most DSM toolsets. In other domains, it may be necessary to use a more powerful language to operate on data exported from the

modeling tool. The most important goal is that the end user should be able to use the code generation simply.

3) Assembling the component library

A domain-specific component library is not always necessary, but it makes the task of code generation development significantly easier. Often, code components already exist from earlier development cycles, at least in the form of reusable pieces of code. Further developing these pieces of code into true components is a relatively easy task for the expert, requiring only the normal developer programming tools. In addition to domain-specific components developed in-house, the library can of course contain generic third-party components.

F. GP - Generative Programming

Generative programming (GP) has a larger scope since it includes automatic configuration and generic techniques, and provides new ways of interacting with the compiler and development environment.

The main *GP* goals are to:

- decrease the conceptual gap between program code and domain concepts (known as achieving high *intentionality*);
- achieve high reusability and adaptability;
- simplify managing many variants of a component; and
- increase efficiency (both in space and execution time).

To meet these goals, GP deploys several principles:

- ***Separation of concerns***: To avoid program code which deals with many issues simultaneously, generative programming aims to separate each issue into a distinct set of code. These pieces of code are combined to generate a needed component.
- ***Parameterization of differences***: As in generic programming, parameterization allows us to compactly represent families of components (i.e. components with many commonalities).
- ***Analysis and modeling of dependencies and interactions***: Not all parameter value combinations are usually valid, and the values of some parameters may imply the values of some other parameters. These dependencies are referred to as *horizontal configuration knowledge*, since they occur between parameters at one level of abstraction.
- ***Separating problem space from solution space***: The problem space consists of the domain-specific abstractions that application programmers would like to interact with, whereas the solution space contains implementation components. Both spaces have different structures and thus we map between them with *vertical configuration knowledge*. The term *vertical* refers to interaction between parameters of two different abstraction levels.
- ***Eliminating overhead and performing domain-specific optimizations***: By generating components statically (at compile time), much of the overhead due to unused code, runtime checks, and unnecessary levels of indirection may be eliminated. Complicated domain-specific optimizations may also be performed.

There are three other programming paradigms which have similar goals to Generative Programming:

- **Generic programming;**
- **Aspect-Oriented Programming** (AOP); and
- **Domain-Specific Languages** (DSLs).

Generic Programming may be summarized as "reuse through parameterization." Generic programming allows components which are extensively customizable, yet retain the efficiency of statically configured code. This technique can eliminate dependencies between types and algorithms that are conceptually not necessary. However, generic programming limits code generation to substituting concrete types for generic type parameters and welding together pre-existing fragments of code in a fixed pattern.

Aspect-Oriented Programming (AOP) Most current programming methods and notations concentrate on finding and composing functional units, which are usually expressed as objects, modules, and procedures. However, several properties such as error handling and synchronization cannot be expressed using current (e.g. OO) notations and languages in a cleanly localized way. Instead, they are expressed by small code fragments scattered throughout several functional components.

AOP decomposes problems into functional units and *aspects* (such as error handling and synchronization). In an AOP system, components and aspects are *woven* together to obtain a system implementation that contains an intertwined mixture of aspects and components. Weaving can be performed at ***compile time*** (e.g. using a compiler or a preprocessor) or at ***runtime*** (e.g. using dynamic reflection).

G. FSP – Framework-Specific Modeling

In an Framework Specific Model (FSM) each concept instance is characterized by a configuration of *features*. Features correspond to *code patterns* that implement them in the application, such as classes implementing framework interfaces, calls to framework methods, and ordering of such calls

A framework provides a set of abstractions, referred to as framework-provided concepts, and means of instantiating them in the framework completion code. The concepts are instantiated by writing the completion code. Object-oriented application frameworks are one of the most effective and widely used software reuse technologies today. The resulting framework completion code implements the difference in functionality between the framework and the desired application.

A Framework-specific modeling languages (FSML) is a Domain-Specific Modeling Language that is designed for a specific framework, called its base framework. A FSML consists of an ***abstract syntax***, a ***mapping of the abstract syntax to the framework API***, and, optionally, a ***concrete syntax***.[11,12,13]

The mapping of the abstract syntax to the framework API defines how concepts and their features map to the framework completion code. The mapping has two parts:

- the *forward mapping*, defining how to generate new code or update existing code for a concept instance; and
- the *reverse mapping*, defining how to recognize an instance of a concept in the code.

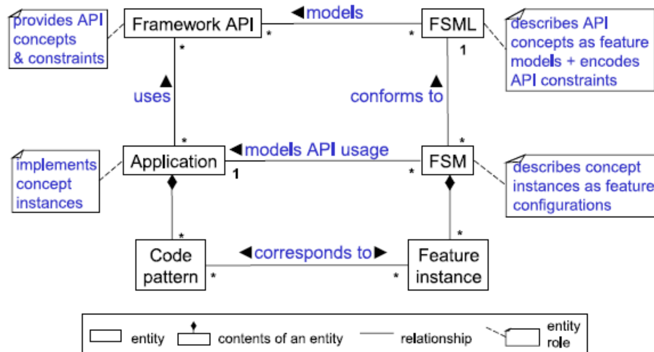

Figure 4. FSM Foundation constructs

The mappings are defined for every concept and every feature individually, allowing for a fine-grained control over mapping execution. The concrete syntax may offer specialized rendering of the models to enhance their comprehension.

The main constructs of the FSM foundation, in the form of collaboration diagram, are shown in Figure 4.

The framework API (implicitly) provides a set of domain-specific concepts along with the constraints on their instantiations. The application code uses the API by implementing instances of these concepts. A concept instance is implemented through code patterns that adhere to the rules and constraints of the API.

Code patterns can be structural (e.g., subclassing a framework class) or behavioral (e.g., calling a framework method in the control flow of an object, order of method calls).

Figure 5. illustrates the three levels of FSML approach.

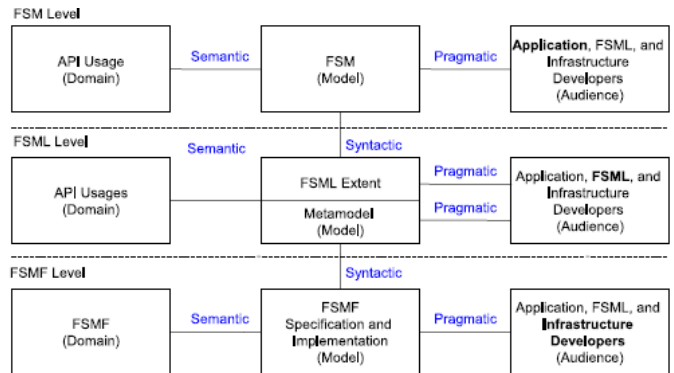

Figure 5. The levels of FSML approach

Object-oriented frameworks are widely used and provide *domain-specific concepts*, which are generic units of functionality.

Eclipse [14] is a universal, open-source platform for building and integrating tools, which is implemented as a set of Java-based object-oriented frameworks. Workbench parts are the basic building blocks of the Eclipse Workbench, which is the working area of an Eclipse user. The parts can interact in various ways, for example, by exchanging events.

H. xUML - Executable UML

Executable UML is at the next higher layer of abstraction of the problem space based on the object-oriented programming language. It bridges the gap between the UML-based design models and the implementation. [15]

The executable models can be compiled or translated to a less abstract programming language, which can be deployed on various platforms for specific implementation. Executable UML [16] means an execution semantics for a subset of actions sufficient for computational completeness. Two basic elements are

required for such subsets: an action language and an operational semantics.

The *action language* specifies the elements that can be used and the operational semantics establishes how the elements can be placed in a model, and how the model can be interpreted.

Executable UML also allow directly executing UML models. It provides an evolutionary model-driven solution to express software. Rather than elaborate an analysis product into a design product and then write code, application developers of executable UML relay on tools to translate abstract application constructs into executable entities. The executable models can be executed given a runtime environment, which also means that they can be validated early in the development lifecycle, as well as be translated to target code achieving near 100% code generation.

Existing executable UMLs can be divided into two kinds. The first kind of executable UML defines an Object Management Group (OMG) action-semantics-compliant language (ASCL) for well-defined, computationally complete formalism. This kind includes executable and translatable UML (xtUML) [17] and xUML [18].

The second kind of executable UML provides action language using simply C, C++, Ada, Java or VBA code. XIS-xModels [19] and Rhapsody [20] provides such executable UML.

I. OMG MDA - The OMG Model Driven Architecture

The Object Management Group™ (OMG™) was formed as a standards organization to help reduce complexity, lower costs, and hasten the introduction of new software applications. One of the major initiatives through which the OMG is accomplishing this goal is by the promotion of Model Driven Architecture® (MDA®) as an architectural framework for software development. This framework is built around a number of detailed OMG specifications, which are widely used by the development community. [7,9]

In 2001 the OMG adopted the Model Driven Architecture as an approach for using models in software development. Its three primary goals are ***portability***, ***interoperability*** and ***reusability*** through architectural separation of concerns.

One fundamental aspect of MDA is its ability to address the complete development lifecycle, covering analysis and design, programming, testing, component assembly as well as deployment and maintenance. [7,9]

The Figure 6. shows the basic MDA environment, according to the OMGs conceptual view. With new platforms and technologies constantly emerging, MDA enables the rapid development of new specifications that leverage them, and streamlines the process of their integration. In this way MDA provides a comprehensive, structured solution for application interoperability and portability into the future. Precise modeling of the solution domain in UML provides the added advantage of capturing its inherent intellectual property in a technology neutral way.

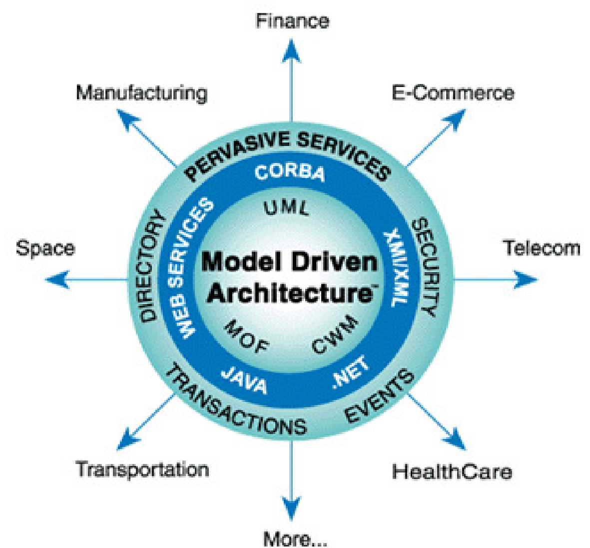

Figure 6. MDA - The Basic Environment

1) MDA – Model Driven Architecture The Major Concepts

In terms of standards, MDA proposes the Meta Object Facility (MOF) [9], a specification that has proven its accuracy in defining the abstract syntaxes of several modeling languages, and that was implemented by several tools.

Major MDA concepts are:

- ***system*** (The context of MDA is the software system, either preexisting or under construction.)
- ***model*** (A model is a formal specification of the function, structure and behavior of a system within a given context, and from a specific point of view.)
- ***model driven*** (Describes an approach to software development whereby models are used as the

primary source for documenting, analyzing, designing, constructing, deploying and maintaining a system.)

- ***architecture*** (The architecture of a system is a specification of the parts and connectors of the system and the rules for the interactions of the parts using the connectors. Within the context of MDA these parts, connectors and rules are expressed via a set of interrelated models.)
- ***viewpoint*** (A viewpoint is an abstraction technique for focusing on a particular set of concerns within a system while suppressing all irrelevant detail. A viewpoint can be represented via one or more models.) MDA specifies three default viewpoints of a system: ***computation independent*** (CI), ***platform independent*** (PI) and a ***platform specific*** (PS).
- ***platform*** (A platform is a set of subsystems and technologies that provide a coherent set of functionality through interfaces and usage patterns.)
- ***platform independency*** (Platform independence is a quality that a model may exhibit when it is expressed independently of the features of another platform.)
- ***platform model*** (A platform model describes a set of technical concepts representing its constituent elements and the services it provides. It also specifies constraints on the use of these elements and services by other parts of the system.)
- ***model transformation*** (Model transformation is the process of converting one model to another within the same system. The transformation combines the platform independent model with additional information to produce a platform specific model.)
- ***implementation*** (An implementation is a specification that provides all the information required to construct a system and to put it into operation.)
- ***MDA models*** (MDA specifies three default models of a system corresponding to the three MDA viewpoints defined above. These models can perhaps more accurately be described as layers of abstraction, since within each of these three layers a set of models can be constructed, each one corresponding to a more focused viewpoint of the system (user interface, information, engineering, architecture, etc.)).
    - ***Computation Independent Model*** (CIM) is also often referred to as a business or domain model because it uses a vocabulary that is familiar to the subject matter experts (SMEs). It presents exactly what the system is expected to do, but hides all information technology related specifications to remain independent of how An MDA mapping provides specifications for how to transform a PIM into a particular PSM. The target platform model determines the nature of the mapping. The CIM plays an important role in bridging the gap which typically exists between these domain experts and the information technologists responsible for implementing the system. In an MDA specification the CIM requirements should be traceable to the PIM and PSM constructs that implement them (and vice-versa).
    - ***Platform Independent Model*** (PIM) exhibits a sufficient degree of independence so as to enable its mapping to one or more platforms. This is commonly achieved by defining a set of services in a way that abstracts out technical details. Other models then specify a realization of these services in a platform specific manner.
    - ***Platform Specific Model*** (PSM) combines the specifications in the PIM with the details required to stipulate how a system uses a particular type of platform. If the PSM does not include all of the details necessary to produce an implementation of that platform it is considered abstract (meaning that it relies on other explicit or implicit models which do contain the necessary details).

J. The MDA Process

Whatever the ultimate target platform may be, the first step when constructing an MDA-based application is to create a platform-independent model expressed via UML.

This general model can then be transformed into one or more specific platforms such as CCM, EJB, .NET, SOAP, etc.

Figure 7. MDA - The transformation of models

A complex system may consist of many interrelated models organized along well defined layers of abstraction, with mappings defined from one set of models into another.

Within this global set of models **horizontal** transformations may occur inside a single layer of

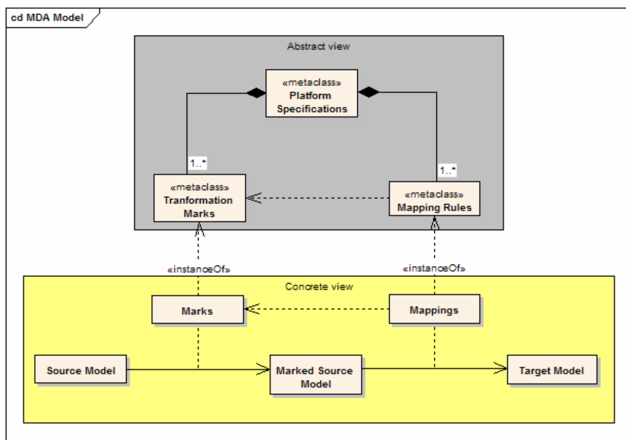

Figure 8. MDA - Model

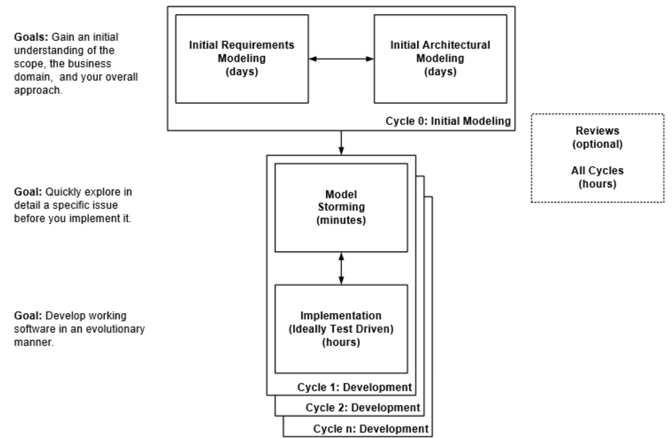

Figure 9. Agile Model Driven Development – Projec Level [26]

abstraction, in addition to the typical **vertical** transformations across layers.

Beyond the perhaps simplistic notion of CIM/PIM/PSM, the two key concepts of MDA are **models** and **transformations**. Figure 7. shows the basic MDA Process model.

Figure 8. illustrates various concepts involved in an MDA transformation for a given platform (i.e. target model) via UML class diagram notation. It distinguishes between the abstract view of the platform's transformation directives, and a concrete implementation in the context of a specific source model to be transformed into this platform.

K. Agile Model Driven Architecture (AMDA)

As the name implies, AMDD is the agile version of Model Driven Development (MDD). An agile MDA process [21] applies the main Agile Alliance principles (e.g. testing first, immediate execution [22, 23]) into a classical MDA process [24, 25].

The difference with AMDD is that instead of creating extensive models before writing source code you instead create agile models which are just barely good enough that drive your overall development efforts.

AMDD is a critical strategy for scaling agile software development beyond the small, co-located team approach that we saw during the first stage of agile adoption.[26]

Figure 9. depicts a high-level lifecycle for AMDD for the release of a system. The envisioning includes two main sub-activities, initial requirements envisioning and initial architecture envisioning.

These are done during iteration 0, iteration being another term for cycle or sprint.

"Iteration 0" is a common term for the first iteration before you start into development iterations, which are iterations one and beyond (for that release). The other activities: iteration modeling, model storming, reviews, and implementation potentially occur during any iteration, including iteration 0.

The time indicated in each box represents the length of an average session.

III. The Current MDE Techniques and Their Limitations

The currently existing techniques that address the reusability of assets provided by methodologists are defined either at metamodeling or model transformation level.

One such technique is the basic ***package dependency***. This relationship enables methodologists to reuse concepts defined by other metamodels when defining new metamodels. Additional OCL constraints [23] may be used as well in order to better tailor the imported package to the exact needs. However the package dependency technique requires the existence of packages of such reusable metamodels.

Another technique is ***profiling***, [32,33] which allows an external asset, also referred to as a *profile*, to extend a given metamodel for storing new information in the conforming models. The profiling technique is mainly based on the principles of branding (i.e., stereotypes) and associated key/value pairs (i.e., tags). It provides methodologists with the possibility to enhance metamodels independently from the methodology.

Model transformations are supposed to be reusable due to the ***interoperability*** that *MOF QVT* provides. Since all transformation languages are supposed to have

a common minimal core at abstract syntax level, a transformation may invoke, or even extend, another transformation. This specification provides the architecture, languages, operational mappings, and core language for the MOF 2.0 Query, View, and Transformation (QVT) specification. The specification defines three related transformation languages: Relations, Operational Mappings, and Core...

QVT language conformance is specified along two orthogonal dimensions: the *language dimension* and the *interoperability dimension*. Each dimension specifies a set of named levels. Each intersection of the levels of the two dimensions specifies a valid QVT conformance point. All conformance points are valid by themselves, which implies that *there is no general notion of "QVT conformance*." Instead, a tool shall state which conformance points it implements.

The declarative parts of this specification are structured into a two-layer architecture (Figure 9.).

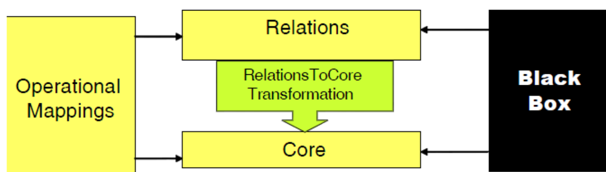

Figure 9. MDA – QVT Declarative Part

The layers are:

- A user-friendly *Relations* metamodel and language that supports complex object pattern matching and object template creation. Traces between model elements involved in a transformation are created implicitly.
- A *Core* metamodel and language defined using minimal extensions to EMOF and OCL. All trace classes are explicitly defined as MOF models, and trace instance creation and deletion is defined in the same way as the creation and deletion of any other object.

Another important technique to be taken into account is ***semantically rich metamodels***, i.e., metamodels that come along with specific support to address an issue that has been neglected till then.

IV. MDDS – CODE GENERATION

The realization of model-driven software development requires effective techniques for implementing code generators for domain-specific languages. [29,34,35]

The core technique is *code generation by model transformation*, that is, the generation of a structured representation (model) of the target program instead of plain text.

This approach enables the transformation of code after generation, which in turn enables the extension of the target language with features that allow better modularity in code generation rules.

The technique can also be applied to 'internal code generation' for the translation of high-level extensions of a DSL to lower-level constructs within the same DSL using model-to-model transformations.

V. INSTEAD OF CONCLUSSION - THE IMPORTANCE OF MENTAL MODELS CREATION

Designing something requires the complete understand what a stakeholder wants to get done. Where support and behavior are aligned, you have a solution. Where a behavior is not supported, you have an opportunity to explore further.[40]

Empathy with a person is distinct from studying how a person *uses* something. Empathy extends to knowing what the person wants to accomplish regardless of whether he/she has or is aware of the thing that is being are designed. It is essential to know the person's goals and what procedure and philosophy she follows to accomplish them.[40]

Mental models give you a deep understanding of people's motivations and thought-processes, along with the emotional and philosophical landscape in which they are operating.

To create a mental model, you talk to people about what they're doing, look for patterns, and organize those patterns from the bottom up into a model.[40]

The three main reasons that describe the advantages of mental models are:

- ***Confidence*** in Your Design — guide the design of the solution;
- ***Clarity*** in Direction — make good user and business decisions;
- ***Continuity*** of Strategy — ensure longevity of vision and opportunity.

The mental model method is a qualitative approach based on interpretation of data that looks like a scientific method. It is a hybrid produced by science and intuition; it's a little of both. It is a very successful method in environments where people are looking to support decisions with real data.

It is also enormously useful in environments where teams can define and communicate product/information design with more intuitive techniques.

REFERENCES

[1] Rakesh Radhakrishan, Mike Wokey, "*Model Driven Architecture Enabling Service Oriented Architecture*", Sun Microsystems, 2004.

[2] *Markus Völter, Jorn Bettin,* Patterns for Model-Driven Software-Development, Version 1.4, May 10, 2004

[3] Alistair Cockburn,"*The End of Software Engineering and The Start of Economic-Cooperative Gaming*", ComSIS Vol. 1, No. 1, February 2004

[4] Tony Clark, Andy Evans, Paul Sammut, James Willans "Applied Metamodelling A Foundation for Language Driven Development, @Xactium, 2004.

[5] *Stephen J.Mellor, Kendall Scot, Axel Uhl, Dirk Weise "MDA Destilled: Principles of Model/Driven Architecture",* Addison Wesley, March 03 2004, ISBN: 0-201-78891-8

[6] Preiss, O. ":*Foundations of Systems and Properties: Methodological Support for Modeling Properties of Software-Intensive Systems*" (Doctoral dissertation, University of Colorado). (2004).

[7] Kleppe, A. G., Warmer, J., Bast, W., "*Explained, M. D. A. The model driven architecture: practice and promise*." (2003).

[8] Marjan Mernik, Jan Heering, Anthonz M. Sloane, "*When and How to Develop Domain-Specific Language"s*, ACM Computing Surveys, Vol. 37, No. 4, December 2005, pp. 316–344.

[9] Mellor,S.J. at.all. "*MDA distilled: principles of model-driven architecture*." Addison-Wesley Professional, 2004

[10] Markus Voelter, "*From Programming To Modeling – and back again"* http://voelter.de/data/articles/FromProgrammingToModeling-1.2-final.pdf, DOI: 10.2298/CSIS110112010P

[11] Micha l Antkiewicz and Krzysztof Czarnecki, Framework-Specific Modeling Languages with Round-Trip Engineering

[12] OMG: UML 2.0 Super Structure Specification. http://www.omg.org/cgi-bin/doc?ptc/2004-10-02

[13] Mellor, S.J., Balcer, M.J.: Executable UML: A Foundation for Model Driven Architecture. Addison Wesley Professional, 2002

[14] Eclipse Foundation: Eclipse. http://www.eclipse.org/ (2006)

[15] Chriss Raistrick, Paul Francis, John Wright, Colin Carter, Ian Wilkie "*Model Driven Architecture with Executable UML*"

[16] Object Management Group, UML 2.0 Superstructure Final Adopted specification, ptc/03- 08-02, August 2003.

[17] MentorGraphics,BridgePointDevelopmentSuite,http://www.mentor.com/products/embed-ded_software/nucleus_modeling/index.cfm, 2007.

[18] Kennedy Carter: Executable UML (xUML), http://www.kc.com/xuml.php, 2007.

[19] Luz, M.P., da Silva, A.R.: Executing UML Models. 3rd Workshop in Software Model Engineering (WiSME 2004), IEEE Computer Society, Lisbon, Portugal, (2004)

[20] Gery, E., Harel, D., Palachi, E.: Rhapsody: A Complete Life-Cycle Model-Based Development System. In: Third International Conference on Integrated Formal Methods (IFM)(2002)

[21] [A18] Stephen J. Mellor. Agile mda. Technical report, Project Technology, Inc., 2005.

[22] Scott W. Ambler. Agile Model Driven Development (AMDD).

[23] Kent Beck. Test-Driven Development By Example. Addison Wesley, 2002.

[24] Object Management Group. MDA Guide Version 1.0.1.http://www.omg.org/docs/omg/03-06-01.pdf, 2003.

[25] Susumu Hayashi et al. "Test driven development of uml models with smart modeling system." In Lecture Notes in Computer Science, volume 3273, pages 395{409, 2004.

[26] http://www.agilemodeling.com/essays/amdd.htm, 2007.

[27] Zef Hemel, Lennart C. L. Kats, Danny M. Groenewegen, Eelco Visser "*Code generation by model transformation: a case study in transformation modularity*", Softw Syst Model (2010) 9:375–402, DOI 10.1007/s10270-009-0136-1

[28] Kats, L.C.L., Kalleberg, K.T., Visser, E.: "*Domain-specific languages for composable editor plugins*." In: Ekman, T., Vinju, J.,(eds.) Proceedings of the Ninth Workshop on Language Descriptions, Tools, and Applications (LDTA 2009), Electronic Notes in Theoretical Computer Science. Elsevier, Amsterdam, 2009

[29] Kelly, S., Tolvanen, J.-P.: "*Domain-Specific Modeling. Enabling Full Code Generation*." Wiley, New York (2008)

[30] D. Spinellis, "*Notable Design Patterns for Domain Specific Languages*," *J. Systems and Software,* vol. 56, no. 1, 2001, pp. 91–99.

[31] T. Stahl, M. Völter, "Model-Driven Software Development:Tehnology, Engineering, Management", John Wiley & Sons, Ltd.2006.

[32] Gordana Milosavljevic, "*Improving Methods of Rapid Development of Adaptive Business Information Systems*", Doctoral dissertation, Novi Sad, Faculty of Technical Sciences, Trg Dositeja Obradovića 6, 2010

[33] Branko Perisic, Gordana Milosavljevic, Igor Dejanovic, Branko Milosavljevic,"*UML Profile for Specifying User Interfaces of Business Applications*",ComSIS Vol. 8, No. 2, Special Issue, May 2011,pp. 405-426

[34] I. Dejanović, „*Metamodel, editor modela i generator poslovnih aplikacija*", Magistarska teza, Fakultet tehničkih nauka, Novi Sad, 2008.

[35] I. Dejanović, G. Milosavljević, B. Perišić, M. Tumbas, "*A Domain-Specific Language for Defining Static Structure of Database Applications*", Computer Science and Information Systems, Jun 2010.

[36] MIC Model Integrated Computing, Institute for Software Integrated Systens, http://www.isis.vanderbilt.edu/research/MIC, accessed on february 2014.

[37] Benoît Langlois, Consuela-Elena Jitia, Eric Jouenne, "*DSL Classification*",

[38] David H. Lorenz. Boaz Rosenan," A *Comparative Case Study of Code Reuse With Language Oriented Programming*", arXiv:1103.5901v1 [cs.SE] 30 Mar 2011

[39] Markus Voelter," *Embedded Software Development with Projectional Language Workbenches*", Proceedings of MODELS 2010 and at http://www.voelter.de/data/pub/Voelter

[40] [Men29]Indi Young, *"Mental Models: Aligning Design Strategy with Human Behavior",* Rosenfeld Media, LLC 705 Carroll Street, #2L Brooklyn, New York 11215 USA,2008