Magisterarbeit

Titel der Magisterarbeit

Model Driven Configuration Management

Verfasser

Bakk. Walter Pindhofer

angestrebter akademischer Grad

Magister der Sozial- und Wirtschaftswissenschaften (Mag. rer. soc. oec.)

Wien, März 2009

Studienkennzahl lt. Studienblatt:	A 066 926
Studienrichtung lt. Studienblatt:	Wirtschaftsinformatik
Betreuer:	Prof. Dr. Stefan Biffl

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Wien, März 2009 Walter Pindhofer

Persönliches

Name:	Bakk. Walter Pindhofer
Geburtsdatum:	21.10.1982
Ausbildung:	10/02 - 03/06 Bakkalaureatsstudium Wirtschaftsinformatik, TU Wien
	Thema der Abschlussarbeit: Workflow based Software Engineering
	09/96 - 06/01 HTL, Elektronische Datenverarbeitung und Organisation
Beruf:	Software Entwickler, Raiffeisen Zentral Bank

Abstract

Model Driven Configuration Management (MDCM) deals with the concept of representing configurations and software development architectures using models based on common modeling languages like UML. These models can be used as an input for configuration management (CM) suites, as a foundation for an Model Driven Software Development (MDSD) workflow or to generate configuration files for different purposes. As a result of merging CM and MDSD, the target audience ranges from software developer to configuration manager. Even though there is a brief explanation of all used technologies, previous knowledge in CM, MDSD and UML will help to understand the covered topics. Each role mentioned above has to fight with the usual problems of software development: redundancy and error-proneness, inconsistency and the resulting higher amount of work.

The main approach of MDCM is to use primary general models as a foundation for generating other more detailed models by using transformations. This procedure is also used by MDSD, but MDCM enhances this approach with models describing configurations. Based on a primary, platform independent model, a model for configurations and software development can be generated. The model for configuration management includes information on the configuration items and their relations and the model for software development defines the software components (classes, packages and methods). All the needed functionality like writing and transforming the models, is provided by the Model Driven Configuration Editor (MDCE), which is one big achievement of this work. It offers easy modelling of the configuration items and software components as well as the generation of models and application skeletons. This approach reduces redundancy and increases the cooperation between development and configuration management.

MDCM and the editor support the software architect on creating the models for the development system and on coordinating those concepts with the configuration manager. The latter will benefit from the import of the configuration models to the CM suite provided by MDCE. And the software developer can use the code generation and MDSD functionality of MDCE to automatise parts of his work.

Criteria for a good solution contain less redundancy, a procedure model for the development processes and tool support for all workflows, which should be easy to use and individually extendable. This work tries to accomplish these targets through the following achievements:

- a concept for merging CM and MDSD called MDCM
- a process model for MDCM
- the model driven configuration editor MDCE
- instructions how to create and enhance the MDCE

Kurzfassung

Model Driven Configuration Management (MDCM) ist ein neuartiger Ansatz, sowohl Konfigurationen als auch Architekturen der Softwareentwicklung durch gemeinsame Modelle darzustellen. Diese Modelle sollen dann als Input für Konfigurationsmanagement (CM) Tools und als Basis für die modellgetriebene Generierung von Code oder Konfigurationsdateien verwendet werden können. Durch die Verschmelzung von Konfigurationsmanagement und modellgetriebener Softwareentwicklung (MDSD) reicht das Zielpublikum dieser Arbeit von Konfigurations-Managern bis hin zu Software-Architekten und Entwicklern. Alle diese Rollen haben mit denselben Problemen zu kämpfen: Redundanz und die daraus entstehende Fehleranfälligkeit und Inkonsistenz beziehungsweise die dadurch zusätzlich anfallende Arbeit.

MDCM verwendet allgemeine plattform-unabhängige Modelle, um ein System zu beschreiben. Aus diesen Modellen werden dann über Transformatoren detaillierte Modelle generiert. Dieses Vorgehen kennt man bereits vom MDSD. MDCM erweitert dieses Vorgehen um Modelle, die Konfigurationen beschreiben. Dadurch kann das allgemeine Modell sowohl als Basis für plattform-spezifische Modelle als auch für konfigurations-spezifische Modelle verwendet werden. Das Konfigurations-Modell beschreibt die Konfigurations-Einheiten und deren Beziehungen, während das plattform-spezifische Modell die Software-Komponenten und deren Verbindungen beschreibt. Der Model Driven Configuration Editor (MDCE) wurde im Rahmen dieser Arbeit entwickelt, um das Arbeiten mit und Generieren von Modellen zu vereinfachen. Der MDCE ermöglicht es, fertige Konfigurationen sowie das Grundgerüst von Applikationen zu generieren.

Mit Hilfe von MDSD und auch des Editors soll die Zusammenarbeit zwischen Entwicklern/Architekten und Konfigurations-Managern verbessert werden. Der Software Architekt soll dabei bei der Erstellung von Modellen unterstützt werden. Außerdem soll die Kommunikation zwischen Entwicklern und Konfigurations-Manager durch die einheitlichen Modelle verbessert werden. Letzterer profitiert außerdem von der Möglichkeit, aus den Modellen fertige Konfigurationen bzw. Input für CM Tools zu generieren.

Kriterien für eine gute Lösung beinhalten die Verminderung von Redundanz im Entwicklungsprozess, ein Vorgehensmodell basierend auf der Idee von MDCM und durchgängigen, flexiblen und erweiterbaren Tool-Support für alle Arbeitsvorgänge. Diese Kriterien sollen durch folgende Entwicklungen erreicht werden:

- MDCM - ein Konzept zur Verschmelzung von Konfigurations-Management und modellgetriebener Softwareentwicklung
- ein Prozessmodell für MDCM
- ein flexibler Ansatz bzw. ein Vorgehensmodell, einen Editor für MDCM zu entwickeln
- der MDCE, ein Prototyp für Tool-Support des MDCM Prozesses.

Contents

1 An Introduction to Model Driven Configuration Management

This chapter explains all the topics of the abstract in greater detail. The context of Model Driven Configuration Management (MDCM) and the target audience are defined, MDCM and the problems addressed by it are described and an approach for providing a good solution for the challenges is introduced.

1.1 Context

The main objective of this work is to connect software development (SD) and configuration management (CM) through universal models. These models describe the target area of a development process and provide the foundation for the SD- and CM-specific models which will be created using the model transformation concept of Model Driven Software Development (MDSD). The SD-specific model provides the input for code generation and the CM-specific model will be transformed into XML files, which define the configuration items and relations for CM tooling or simply serve as configuration files for different applications or technologies.

MDCM offers procedure models, metamodel definitions and best practice methods which altogether should enable the reader to deal successfully with the fusion of SD and CM. As a result of MDCM, the Model Driven Configuration Editor (MDCE) has been created. The MDCE is a graphical editor in Eclipse to create, modify and transform XML configurations. MDCE uses the Eclipse Modeling Framework (EMF) and the Graphical Modeling Framework (GMF) as foundation for visualisation and modeling and openArchitectureWare (oAW) for the transformation. MDCE serves as a tooling for MDCM. It creates the implementation for SD as well as the XML input for CM tools. This work provides a detailed step-by-step manual to easily develop a custom MDCE for a project. Figure 1.1 shows MDCM and MDCE and their relation to software disciplines, tools and languages.

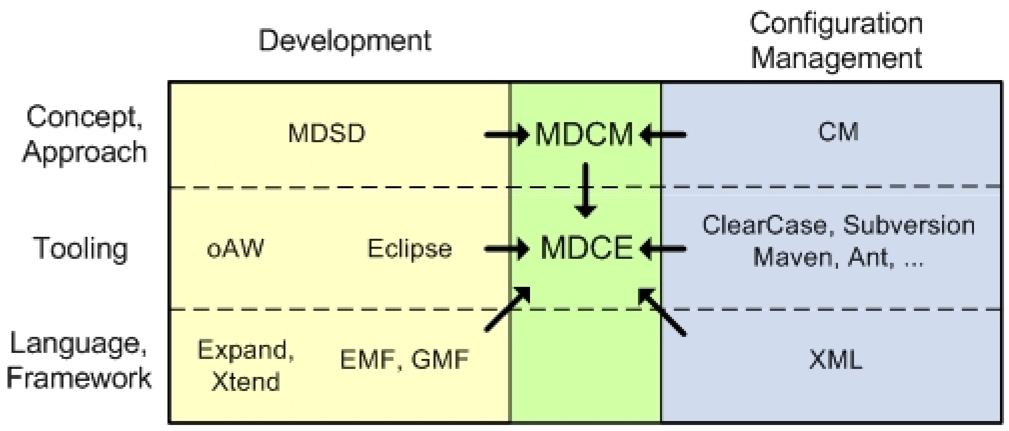


Figure 1.1: Context of Model Driven Configuration Management
(MDCM is a consolidation of the ideas behind MDSD and CM. The tooling for MDCM, the MDCE is using Eclipse and its Plug-Ins and the MDSD Framework oAW as well as XML.)

1.2 Target audience

As a result of the interdisciplinary nature, this work involves many different roles and also demands comprehensive knowledge. MDCM delivers interesting new approaches for project and configuration managers, developers and designers. These roles and their relation to MDCM will be described in section 1.5 Roles. To understand the concepts of this work, knowledge in (model driven) software development, configuration management, Java, Eclipse, XML and XML Schema is required.

In chapter 3 Related work and applied technologies, all MDCM essential technologies are explained and references for more detailed information are provided.

1.3 Challenges and problems

Each role mentioned above has to fight with the usual problems of a large software development process. Let's start with the configuration manager. In most cases, he will have to model and document his configurations before managing them. Therefore he writes for example a model defining the client/server environment or a software component model. After modeling the configurations, he has to include them to CM tooling. This task requires to manually insert the same configuration items as modelled before, a clearly redundant work. Of course the configuration manager is not the only role struggling with redundancy. The software architect has to design UML models for the application area in form and content very similar to some of the software component models written by the configuration manager. These models provide the foundation for the software developer who does nothing else than converting the models to programming code. The obvious redundancy not only brings a higher amount of work along, but also many other accompanying problems. Those issues reach from inconsistency to error-proneness. Some of the problems bothering the described roles result from a insufficient connection between configuration management and software development and some have already been addressed by MDSD.

Another issue addressed by MDCM or more specific by the MDCE is the work with XML based configurations. Although XML provides human readable and an easy to learn structure, it's still very error-prone to create or manipulate XML files in an simple text editor or even with toolings like XML Spy. Therefore, MDCM provides the Model Driven Configuration Editor (MDCE), where it is possible to define the structure and elements of a configuration as metamodels and afterwards use the visual editor of the MDCE to simply create XML configurations by drag and drop. This reduces error-proneness and even allows users unfamiliar with XML to edit those configurations.

1.4 Advancements through Model Driven Configuration Management

In order to reduce the problems of redundancy, a redesign of the software development process is necessary. The idea is to merge some aspects of CM and SD in order to benefit from the similarities between them and to provide an end-to-end tool support. Therefore, MDCM introduces a new process model summarized next.

The architect creates models of the application area. These models will be extended with configuration items and configuration relations for CM on one side and with specific information for development on the other. The CM models will be used to create input for CM tooling and the models for development will be transformed into a skeletal implementation including packages, classes, methods and attributes. This approach reduces some of the redundant work and integrates CM and SD better to the whole software process. Figure 1.2 illustrates the different steps and artefacts of the MDCM process.

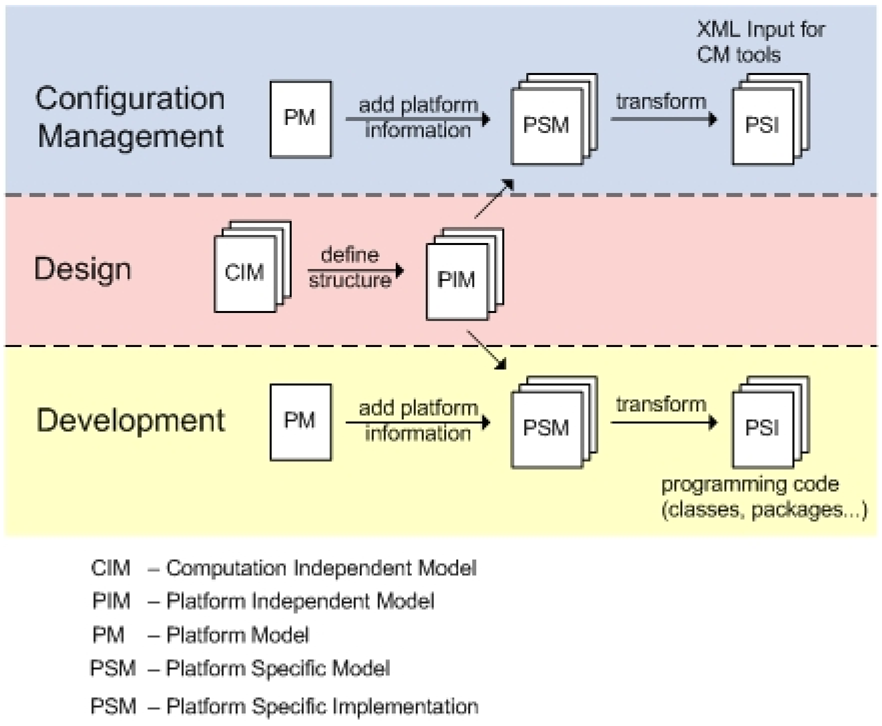


Figure 1.2: Model Driven Configuration Management process artefacts
(The different artefacts of MDCM and are arranged in layers, according to the management discipline they belong to.)

1.5 Roles

Each role has specific activities and responsibilities throughout the MDCM process. This section introduces the main roles addressed by MDCM.

Project manager

The project manager is responsible for doing software engineering using the MDCM approach. He has to assure that the MDCM process is running and that everyone in the team is living the ideas of MDCM, which is very important because MDCM provides a whole new concept for doing development and configuration management. Since most of the considerations regard management aspects and strategic decisions, it might be the role of the project manager which benefits the most of reading this work.

Architect or designer

The software architect has to design and specify the application using the models and rules of MDCM. He has to design and plan the transformations between the models and he has to collaborate with the configuration manager and developer in order to match his designs with their requirements. He also has to decide, which configurations of the application have to be managed by the MDCE. MDCM offers metamodels and editors which should be interesting and helpful for architects.

Configuration manager

The configuration manager is responsible for the design and setup of software, hardware and system configurations. He has to enrich the models of the architect with configuration specific informations. As a result, MDCM delivers a file containing XML configuration item definitions, which has to be imported into the used CM tooling. For a configuration manager, especially the aspect of modeling configurations and transform those models to XML should be interesting.

Developer

In MDCM the developer has to support the architect and configuration manager by developing the defined transformations between the different models. MDCM only provides guidelines and rules for transformation, but the actual transformation has to be adapted for the special needs of each project. The developer is also responsible for the code generation and of course the coding itself.

1.6 Criteria for a good solution

The strategy of merging configuration management and software engineering can be pursued in many ways. But at least a process following such a strategy should be more efficient than before. Therefore, the next list shows some criteria for a good solution:

- Easy access and fast results through a well defined process including the communication between the roles.
- MDCM should increase the collaboration between configuration management and software engineering and between the roles architect, configuration manager and developer and should lead to a more economonic communication.
- An efficient and continuous tool support has to be provided.
- The tooling should be flexible and extendable.
- Configurations of the target application will be easier through MDCM

But the most important criteria and the initial part of all considerations is to **decrease the work amount** through **decreasing redundancy**.

1.7 solution approach

A widespread concept has been created based on the criteria for a good solution above. When creating this concept, it became obvious that there are two critical factors for success:

- a comprehensive process model supported by
- a flexible and expandable tooling

In order to achieve these goals, a procedure model and some best practice methods based on existing approaches for configuration management and software engineering have been developed. The following list shows the major steps of the MDCM procedure model.

- general models are created based on the MDCM metamodels
- through transformation, these models are enriched with platform specific information
- the configuration management specific models are used to generate input for CM tooling
- the software engineering specific models are used to generate classes, packages and methods
- models are used to generate application configuration files.

The whole process uses many methods and approaches of the Model Driven Architecture (MDA). In order to get an end-to-end, flexible tool support, a manual describing how to fast and easy build an MDCE is one part of the MDCM. Because MDCM doesn't provide an editor, but rather offers a manual describing how to create an editor, it is exceptionally flexible and the editor can be adapted for every use. Additionally there are different examples which show already built MDCEs.

2 From traditional Software Development to Model Driven Configuration Management

This chapter describes the benefit of using Model Driven Configuration Management (MDCM) and its advantages to the traditional software development process. It shows the underlying motivation behind MDCM and many reasons, why MDCM has to be applied to new projects. This chapter should also motivate to read more of the work and to get further involved in MDCM.

The next three sections describe the different software development processes and their connections to configuration management. This examination has a special focus on the artefacts of the processes and the level of automation between the different iterations and phases. The last of the three sections describes MDCM and also summarizes the main targets and motivation behind it.

2.1 The traditional Software Development process with Configuration Management

Traditional software development processes usually have the following 5 major phases, which have their origin in the waterfall model [ROYCE 1987]:

- **Analysis (Requirements):** Models the target system with focus on the challenges of the system and the requirements of the customer. The analysis is not concerned with the concrete realisation. Additional interesting informations on software analysis are provided by [JACKSON and RINARD 2000].
- **Design:** Based on the requirements, the structure, behaviour, usage and coverage of the software implementation are specified.
- **Implementation:** The goal of the implementation phase is to build an application based on the design documents which meets all defined requirements.
- **Test:** The application will be verified against the functional and non-functional requirements and its specification.

- **Deployment:** Install, set-up and run the application on the target system.

Analysis, design and implementation are only very loosely coupled at traditional software development. The artefacts of the different phases have no direct connection and the divisions work as far as possible autonomous. This method of operation leads to fast and independent specifications, but at the expense of high redundant work and suboptimal communication. Traditional software development (SD) does not define concrete directives or rules for the design and requirement documents.

All phases of the SD process will be supported by configuration management. The different configurations have to be created and the configuration items will be identified and added without or with only minimal support of the SD teams. So configuration management (CM) offers functionality for SD, but the connection from SD to CM is neither defined nor really efficient. Figure 2.1 shows the traditional SD process, its artefacts and the relationship to CM.

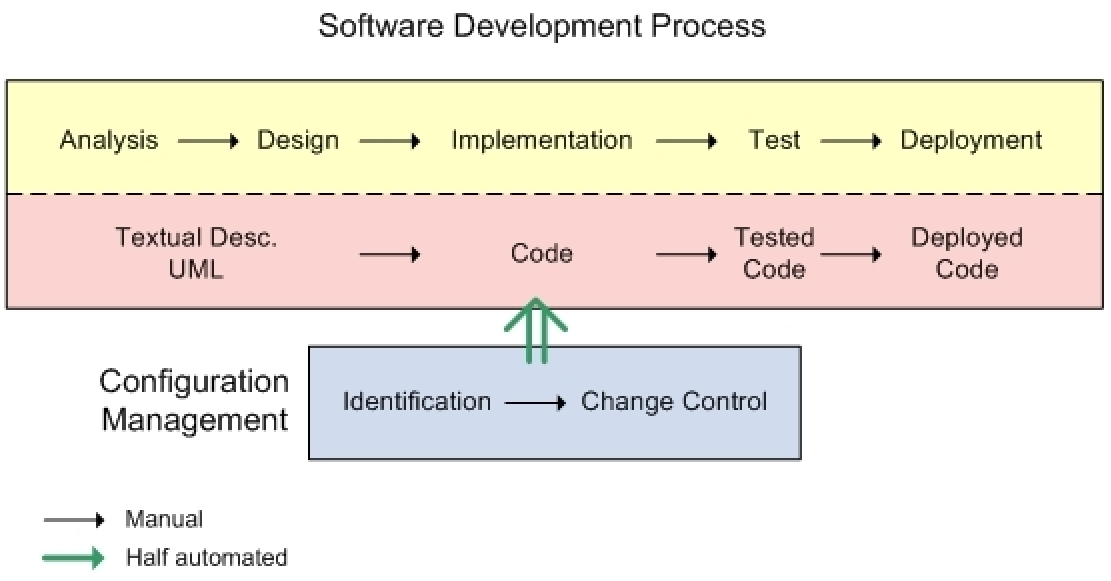


Figure 2.1: The traditional Software Development process with Configuration Management (In this process, each step has to be done manually, only the configuration management is half automated because of tooling support)

2.2 The Model Driven Software Development based process and Configuration Management

Model driven software development(MDSD uses the same procedure models as traditional software development does. MDSD also follows the 5 phases explained in the previous section, but unlike traditional SD, MDSD mostly uses models to describe the target system and only uses textual descriptions as a compromise. The models of the different phases are connected and can be partially or completely generated automatically (just as well as the implementation). Additional informations on Model Driven Architecture (MDA) and MDSD can be found in section 3.2 Model Driven Software Development.

Some of the redundant work will cease to exist by the use of the automatization of MDSD. However the modeling is more complex and time-consuming. Because the models have to have a specific structure and more informations than the SD models in order to make the automatization possible.

Figure 2.2 illustrates the MDSD process and the connection to CM. The integration of CM in MDSD is unfortunately as insufficient as the integration in traditional SD. As can be seen, the main differences between the two approaches are the artefacts. When using MDSD, the artefacts are mostly models defined upon a specified structure which enables transformations and code-generation.

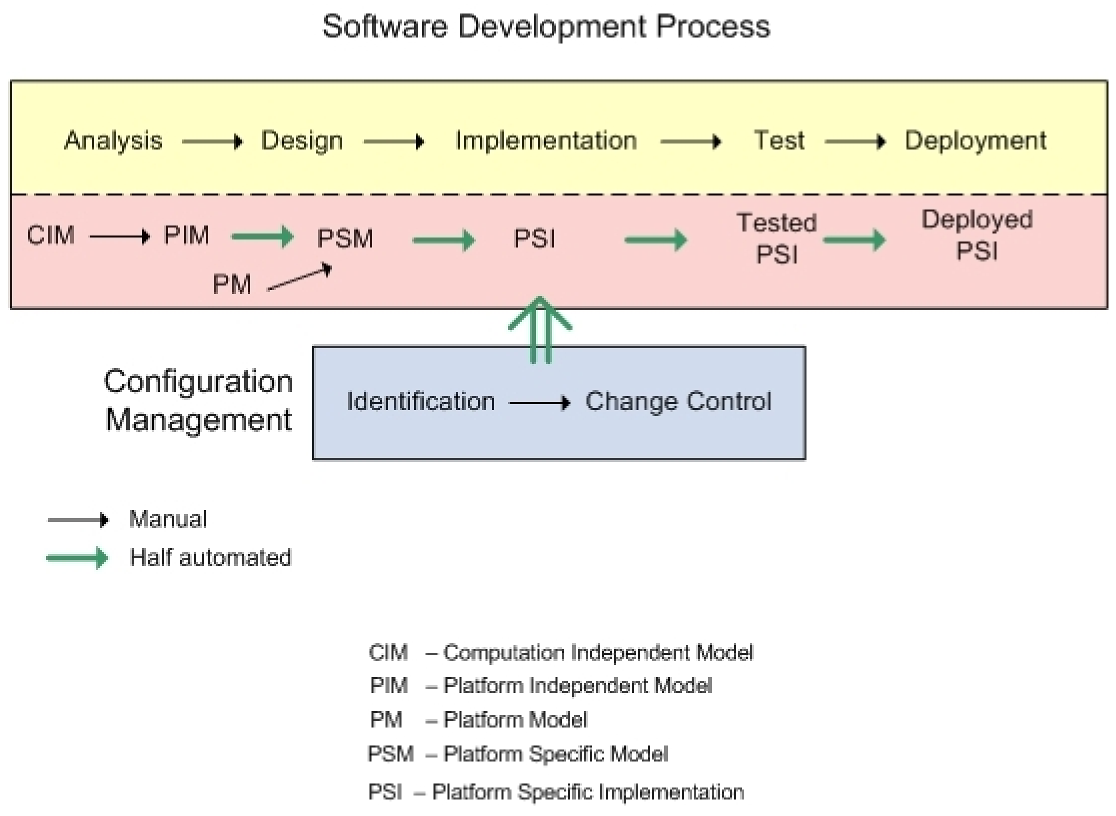


Figure 2.2: A Model Driven Software Development process with Configuration Management (In MDSD, the mapping between the development specific artefacts is automated through transformations. But still the documentation has to be created and updated manually)

2.3 The Model Driven Configuration Management process

MDCM extends the MDSD approach by directly integrating parts of CM into the SD process. Figure 2.3 illustrates the MDCM process.

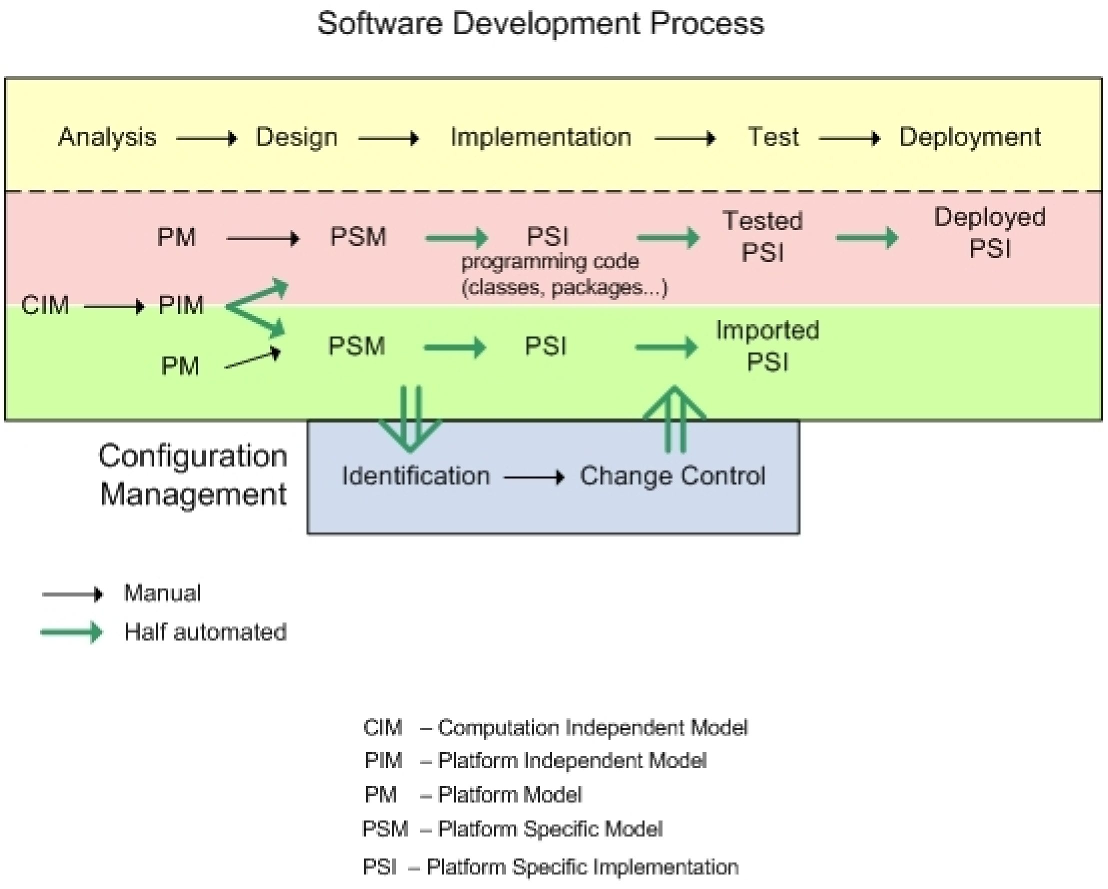

Software Development Process
Analysis
Design
Implementation
Test
Deployment
PM
PSM
PSI
programming code (classes, packages...)
Tested PSI
Deployed PSI
CIM
PIM
PM
PSM
PSI
Imported PSI
Configuration Management
Identification
Change Control
Manual
Half automated
CIM – Computation Independent Model
PIM – Platform Independent Model
PM – Platform Model
PSM – Platform Specific Model
PSI – Platform Specific Implementation

Figure 2.3: The Model Driven Configuration Management process
(Yellow - Phases of the Development Process;Red - MDSD Artefacts;Green - CM Artefacts
By adding CM focused artefacts to the model driven approach, a better cooperation between configuration management and software development is possible and the redundancy will be reduced.)

In MDCM, CM models are specified like and connected with MDSD models. Therefore development specific CM models can be linked with MDSD models and configuration items and their dependencies can be identified more easily or even automatically. Even the input for CM tooling can be generated and imported. The next list summarizes the targets and the advantages of MDCM.

Targets of MDCM

- **Decrease redundancy** in order to also decrease the accompanying problems inconsistency, error-proneness and the additional amount of work.
- **Integrate CM to SD** in order to benefit from the similarity of the SD related configuration tasks and also to increase the quality of CM through close collaboration.
- **Flexible, extendable and consistent tool support throughout the whole SD process** in order to optimize the SD and CM process and to offer the most efficient working environment.

3 Related work and applied technologies

The foundation for Model Driven Configuration Management (MDCM) is provided by known technologies and popular approaches. MDCM uses the Model Driven Software Development (MDSD) approach to bring configuration management closer to software development. Section 3.1 specifies the meanings of configuration management (CM) in the context of MDCM and section 3.2 describes MDSD and also Model Driven Architecture (MDA).

As an important part of MDCM, the Model Driven Configuration Editor (MDCE) provides a continuous tool support. MDCE is based upon the Graphical Modeling Framework (GMF) and the Eclipse Modeling Framework (EMF), two Eclipse frameworks which in combination offer efficient graphical modeling. Section 3.3 is all about these two Eclipse frameworks. Transformations and generation of models and code is a very relevant part of MDCE. It uses the openArchitectureWare framework and as additional support XML transformations through JDOM. Section 3.4 will provide all necessary information about transformations in MDCM.

3.1 Configuration Management

This section provides an overview of the fields of activities of configuration management (CM), the different concepts, tools and approaches of CM including the most interesting part, software configuration management (SCM). CM is such a big issue that there exist hundreds of different definitions and approaches concerning this topic. Since CM is enhanced constantly and is therefore still growing, it's also really hard to find a book or work which includes all the aspects of CM. Configuration management is the foundation of this whole master thesis. Therefore it is indispensable to exactly define the meaning of the term CM in the context of this work. The definitions and explanations in this section cannot be seen as complete or unalterable. They have to be seen as the starting point of MDCM and its results. CM can be done like described in the next sections, but there are plenty of other ways to perform CM. The next two definitions show, how much CM has really changed in the last two decades.

> **1978, Definition Configuration Management:** CM is generally concerned with the consistent labeling, tracking, and change control of the hardware elements of a system. [Bersoff et al. 1978]

> **2003, Definition Software Configuration Management:** Current definition would say that SCM is the control of the evolution of complex systems. More pragmatically, it is the discipline that enables us to keep evolving software products under control, and thus contributes to satisfying quality and delay constraints. [Estublier 2000]

When asking a couple of software project managers or developers, how they would describe CM, it can be expected that the answers will differ very much. Even the position of change management to configuration management isn't clear. Some say it's a management approach very similar to CM, other say it enhances CM. In this work, change management is considered to be an important part of configuration management. One good and often used definition of CM is the one by IEEE. Even it's a bit out of date, it summarizes the essential parts of CM.

> IEEE Std-729-1983 Definition Configuration Management
>
> Configuration management is the process of identifying and defining the items in the system, controlling the change of these items throughout their lifecycle, recording and reporting the status of items and change requests, and verifying the completeness and correctness of items [1].

Whatever definition is preferred, the really important thing, and that's where everyone will agree, is what CM can do for each particalur business or development process. Used the right way, CM will ease the development, change and release of software, it will help developers building the application and it will simplify communication between the project groups. When a project reaches a certain size, CM is not an additional management discipline to raise the quality, it's way more than that. In really large projects, it's really the only way to be able to get at least any quality. The book [Popp 2008] describes the basic tasks and tooling necessary for the traditional software configuration managenemt. This book is a good introduction on CM and can also be used to apply CM for a concrete project. Nonetheless it does not capture CM in its whole complexity. There are even approaches to coordinate the whole software engineering using configuration management [Grinter 1996].

3.1.1 History of (Software) Configuration Mangement

Configuration management was originally invented to support hardware development and production control. This was in the late 1950s or early 1960s. The first theses on change and configuration control were written at the end of the 60s. At this time, software configuration management was done manually and it was nothing more than CM applied to software development. The first papers dealing with SCM were written at the end of the 70s. These papers were SOFTWARE CONFIGURATION MANAGEMENT by Edward H. Bersoff, Vilas D. Henderson and Stan G. Siegel (1978) and Software configuration management by J.K. Buckle (1982).

[1] from http://www.ieeexplore.ieee.org

These two works introduce some basic concepts like identification of the Configuration Items (CI) and baselines.

It is only after this date, that the use of software tools working directly with software artefacts representing the actual resources, has allowed SCM to grow as an autonomous entity (from traditional CM). From this point, SCM continued enhancing at a great speed. New approaches like process control and release management were added to SCM. The tools supporting SCM were getting more powerful and as the size of many development projects was getting bigger, many project managers discovered the significance of using SCM tools. As time goes by, software development and SCM has changed and is still changing.

> In the early 80s SCM focused in programming in the large (versioning, rebuilding, composition), in the 90s in programming in the many (process support, concurrent engineering), late 90s in programming in the wide (web remote engineering). [ESTUBLIER 2000]

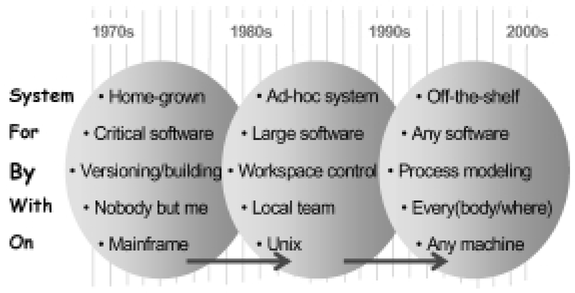


Figure 3.1: Evolution of Software Configuration Management [JACKY ESTUBLIER 2005]

The next two sections describe some of the most important concepts and features of CM and introduce management disciples which are a part of CM.

3.1.2 Concepts of Configuration Management

Although there are many different understandings of CM, the basic concepts remain the same. Everyone doing CM needs methods to identify the configuration items, methods to control the changing and versioning of them and functions to provide tool support.

Configuration items

Configuration items (CI or sometimes also called artefacts) form the basis of configuration management. Each file, requirement or other definable unit which is a part of the CM is represented by a CI. Each CI can be managed and versioned individually. Change management, release management and Versioning, all those parts of CM work with the CIs. When changing a file, a CI is changed and may get a new version number. Usually, when changing a CI, CM has to determine which other CIs are affected. CIs can be related to other CIs in other to get a set of CIs which might be a working configuration or a version of the application. The next table shows a list of CIs, usually managed by CM systems [Jacky Estublier 2005]:

Managed Configuration Items	
System data files	Source code modules
Interface specifications	Requirements specifications
Quality plans	Software development plan
Design specifications	User documentation
Configuration management plans	Test plans
Software architecture specifications	Procedure language descriptions
Test data sets	Test procedures
Test results	Compilers
Debuggers	Linkers and loaders
System build files/scripts	Shell scripts
Operating systems	Other related support tools
Third-party tools	Development procedures and standards

Table 3.1: Configuration Items of Software Configuration Management

Versioning

The first versioning systems (as a part of CM) have been built in the 70s. They already worked very similar to the versioning used by modern change or configuration management approaches. Here is a definition of versioning.

> Versioning concerns the need to maintain a historical archive of a set of artefacts as they undergo a series of changes, and forms the fundamental building block for the entire field of SCM. [Estublier 2000]

Every time a CI (or at this time, simply a file) is changed, a new revision is created. A revision is usually simply displayed by successive numbers. Additionally to this basic feature, new version systems also save the user who has made the changes, the time the file has been changed and the user is able to comment the changes. The function of saving these additional information is called **History**.

Not only a new number is generated, version systems also provide the possibility to reload an older version of the changed CI. Since it is not efficient to save the whole file for each version of a CI, modern version systems only store the differences between the revisions. This approach is called **Deltas**.

Another service provided by most versioning systems is the **multi-user management**. The first approach on multi-user management was the simple check-out/check-in mechanism. Every time a user wants to change a CI, he has to check it out. When a CI has been checked out, no one else is able to change it. When the user has finished the editing, he checks the CI in and the changes are saved. Newer systems allow concurrent editing of files and CIs and merge the changes. This service is also referred to as **Cooperative Work Support**. Patterns for how to do Versioning in a team and in development processes are explained at [BERCZUK 2003]

The Change Control Concept

When developing a complex application, it's usual and necessary to design the application first. Most times a requirements specification, a design document, the implementation and test cases are created. These documents are all related to each other. So if the design has to be changed, the implementation and test cases have to be changed and maybe even the requirements specification as well. Even if the implementation is changed, it may be necessary to change the design document and requirements specification. Often changes of one part of an application affect another part without knowing it. Change control is responsible to control this problem.

When using change control, parts of code cannot simply be changed. Every change has to be registered at the change control system, which opens a change request. Then, all documents which might be affected by the change have to be determined. This can be a very complex job, but with a proper configured system, it's possible to get at least a list of all documents related to the one that has been changed. All documents the change affects have to be checked and altered. After all documents have been modified, the change request is closed. Summarized, the change control follows this process:

- Identify potential change

  Maybe a requirement has to be added or a code fix has to be done. The change has to be noted.

- Analyze change request

  On one side, the technical complexity and possibility of the change request are analyzed. Is it even possible to fulfill the change or is it too difficult to solve? On the other side, the costs, risks and of course the profit of executing the change are analyzed. Only if the profit exceeds the calculated costs and risks, the change request will be accepted. If, for example, a test finds a problem but fixing it is more expensive than the expected profit, the problem won't be solved.

- Evaluate change

  Based on the result of the evaluation, the change request is accepted or declined.

- Implement change

  At first, the requested change is done. After that, all affected CIs have to be modified and therefore the change is propagated to the other system parts influenced by the change. After all changes have been executed, the changes are tested.

- Review and close the request

At last the whole change request is reviewed. It's not enough to do all the changes to call the change request successful. It's important, that the real costs haven't exceed the estimated costs and that the profit is larger than the costs. After the review, the change request is closed.

Building/Rebuilding

At the beginning of software development, building an application out of the source files was not a big thing. One or two compiler commands and the application was running. Nowadays, where huge applications consisting of many different components are built, the building process isn't that easy anymore.

Build tools help to accomplish complex build processes. Such tools are for example Ant or Maven. So build tools are not necessarily a part of CM suites. But since release management is a part of it and the entire source and the needed files are located in the workspace of the CM suite, it is only the next step to include a build tool which works as a part of the CM suite. Most CM suites and tools already have included a build tool.

Workspace Support

As the effort of managing and controlling the software development process raised, many project leaders started to deal with CM. But most developers still didn't see the benefits, CM might be able to provide. Workspace support was the part of CM which was the breakthrough regarding the interest of the developers.

Workspace support simply provides an environment, where developers can work. The workspace is a directory where all necessary files are available. There are two different ways of providing these files. The first way is to link the files directly to the one on the file server. Each time a developer wants to change a file, he has to check it out and after editing it he has to check it in. The file will then be changed on the server. The second and better way, used in most cases is that the files in the workspace are isolated from the other developers, so that the developer can work almost independently. This means, everyone has its own local repository to work with. When changing one or more files, they have to be committed and uploaded to the server. The CM suite is then responsible for keeping the files consistent. Even versioning and the building process can be included in the workspace to ease the daily work.

Product Support

From the beginning, product management was a core functionality of SCM. But product support not only means watching and versioning the many files which constitute the software system. Product support needs to offer much more. The focus here lies on the product as a collection of files. A CM suite should therefore offer a way to define products as a number of artefacts consisting of files. These artefacts have attributes to describe them further and can be related to each other. The sum of artefacts represents a revision of the product. There might be even a lot of different versions of the product, each one is then managed as a sum of artefacts. Product support needs a good building tool supporting the process. That's because the products which are defined here, have to be built and the better the connection between the product support and the building tool is, the easier it is to manage and build the whole product.

3.1.3 Management disciplines of Configuration Management

Configuration management combines many different management disciplines to provide the whole set of support. Each management has a right to exist on its own, but connecting the different approaches to one set will deliver the greatest the benefit. This section describes the different management disciplines combined in configuration management.

Requirements Management and Engineering

A well working and strong requirements engineering (RE) is the key to a successful development process. Without good and clean defined requirements, building an application meeting the expectations of the customer is not possible. Configuration management is important for an effective requirement engineering. CM provides the necessary version and change control and at the best, a CM suite includes a requirements management tool.

There are five areas of RE: Eliciting, documenting and approving the requirements, management of requirements and the traceability of requirements.

Eliciting Requirements

Eliciting the requirements is one of the most critical actions of the whole development process. Hence there has to be enough time to acquire the requirements of a project properly. If the requirements are not articulated right, the project will develop the wrong output and therefore fail. To determine the requirements, a lot of meetings with customers and intern discussion groups have to be arranged.

There are different types of requirements which have to be considered:

- **User Requirements** are the requirements important for the users of the application. These requirements define the behavior and interface of the application and have to be coordinated with the future users.
- **Business Requirements** describe the benefits and costs the customer expects. Business requirements include the estimate project duration, the needed labor and more.
- **Functional Requirements** describe the needed features of the program.
- **Non-Functional Requirements** cover quality aspects like security, performance or multi-platform features.

Document the Requirements

The determined requirements have to be formulated so that they are complete, consistent and coherent. The importance of the requirements is rated and the effort is estimated. All requirements are written down and saved. CM can support the developers and designer by already providing a workspace and versioning. Even the change control can be started at this point.

Approve the Requirements

All requirements which will be implemented have to be approved at this stage of RE. The next step is establishing a set of requirements for the requirements baseline of a given release.

Management of Requirements

The approved requirements have to be managed now. This is done by using the requirements management (RM) process, which reminds on a change control process.

> Typical steps in an RM process include [Moreira 2005]:
> 1. change request gets submitted
> 2. change gets analyzed for impact and effort
> 3. change gets reviewed and ruled on (accept, reject, etc) by the CCB
> 4. change gets assigned to a change agent
> 5. change agent makes change
> 6. change gets validated
> 7. change request gets closed

Traceability of Requirements

The last stage of RM is to build relations between requirements, code and test classes in order to provide traceability for the defined requirements. The benefit of traceability is:

- Only what is defined as required is developed.
- Changes in the code structure can be traced back to the requirements and changes of the requirements can be better adapted to the code.

There are requirement tools for doing most of the described activities. In order to achieve versioning and change control, these tools often have to be combined with a CM tool. A better way to manage the requirements is to use CM tools with integrated requirements management. Many large CM suites already support RM.

Change Management

Change management is the structured approach of establishing and executing change control. Change control is a very time-consuming activity. Not every change has to go through the whole change control process. One task of change management is to decide, which items have to be monitored by change control. This is depending on a few factors:

- **Importance of the file:** The more important a file is for the success of the project, the more likely it has to be added to change control
- **Frequency of changes:** A file which is changed often, like a programming class which is under development, shouldn't be watched by the change control, because the effort is too large.
- **Relations of the file:** The more relations (files that are affected by changes) are existent and the more important those relations are, the more likely it should be added to change control. Because if the file has some important relations, the risk of forgetting to change one related file is too big.

Change management also chooses the used change control tool in order to support the whole control process. It's best to select a CM tool with included change control. And since change control and management is one of the core elements of CM, every good CM tool also supports effective change control.

Another task of change management is to watch over the change control process. The process has to be executed correct and fast. Delays between the steps of the process have to me minimized and employees have to be trained in realizing the change control process the best way.

A good working change management provides fast and exact changes and is most crucial for a successful configuration management. Although it is possible to look at change and configuration management as two different management disciplines, it's more recommendable to see change management as a part of CM [JOERIS 1998].

Release Management

In the last three decades, the structure of applications has changed away from big static architectures to service oriented architectures (SOA) providing reusable services and loosely coupled structures. This and the fact, that less time for design and development cycles is available and the market pressure grows steady, add more and more requirements to the release tasks and therefore to release management. An article describing the basics of release management can be found at [MEHROTRA 2004].

The software release manager

The software release manager is responsible for the whole release process. He has to design and watch over the process of releasing applications. An adequate definition of the role release manager is formulated by Microsoft:

> The release manager is responsible for managing the release process, which includes planning for the release, ensuring user acceptance tests have been completed,

> verifying training has been provided to the affected user community if needed, validating the backout plan, staging the pilot tests, and implementing the full deployment of the release. [MICROSOFT 2004]

The release manager has to care about many things. Some of the challenges facing a software release manager covers the management of the following tasks [LIPIEN et al. 2006]:

- Software Defects
- Issues
- Risks
- Software Change Requests
- New Development Requests (additional features and functions)
- Deployment and Packaging
- New Development Tasks

The release management process

The following picture illustrates the main components of the release management process.

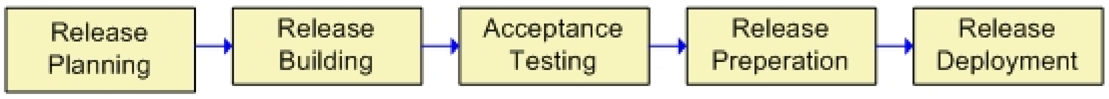


Figure 3.2: Release management process based on [MICROSOFT 2004]

The first step of **release planning** is to identify the resources required to deploy a release into production environment. The structure and components of the release have to be defined, a schedule for the release activities has to be created. Next step is to formulate a release plan which consists of the release activities, their priority and the needed resources to execute the plan. Since the release of an application into production is a critical activity, the release plan also deals with the potential risks of the release. After documenting all the planning actions, the release can be built.

Although the **release building** can be done manually, it is cost and time saving to use tools and technologies supporting the building process. Therefore, the first step is to define those tools. Then a release package is defined, which executes the release. This release package has to be tested before it is performed in the production environment. If these tests lead to the desired result, namely meeting the planned requirements, the release package is submitted.

Now it's time to test the release and the release package in an environment similar to the production environment. This step is not necessary for every release. But the more complex and important the release is, the more needful it is to execute **acceptance testing**. Acceptance testing must then create and design the testing environment and execute the release package. All tests have to be started and evaluated on this system. The results of the test cases are documented and all found problems and errors have to eliminated.

After all this testing and documenting, it's now time to prepare the production environment for the release. The release manager and the head of the production have to assure, that all needed resources are available, all affected users and developers have been informed and all related changes have been forwarded to the change manager. **Release preparation** is the very last step before the actual release is done. Hence, all set up activities have to be completed.

Before the actual release can be done, the deployment group for the release has to be selected and all affected people have to be informed one more time. Then the **release deployment** is executed. After the successful deployment, the release has to be checked and tested. In the next few weeks, the feedback of the users working with the new release has to be processed and the requested changes have to be done. When everything is working fine, the release process can be closed.

3.1.4 Parts of Configuration Management

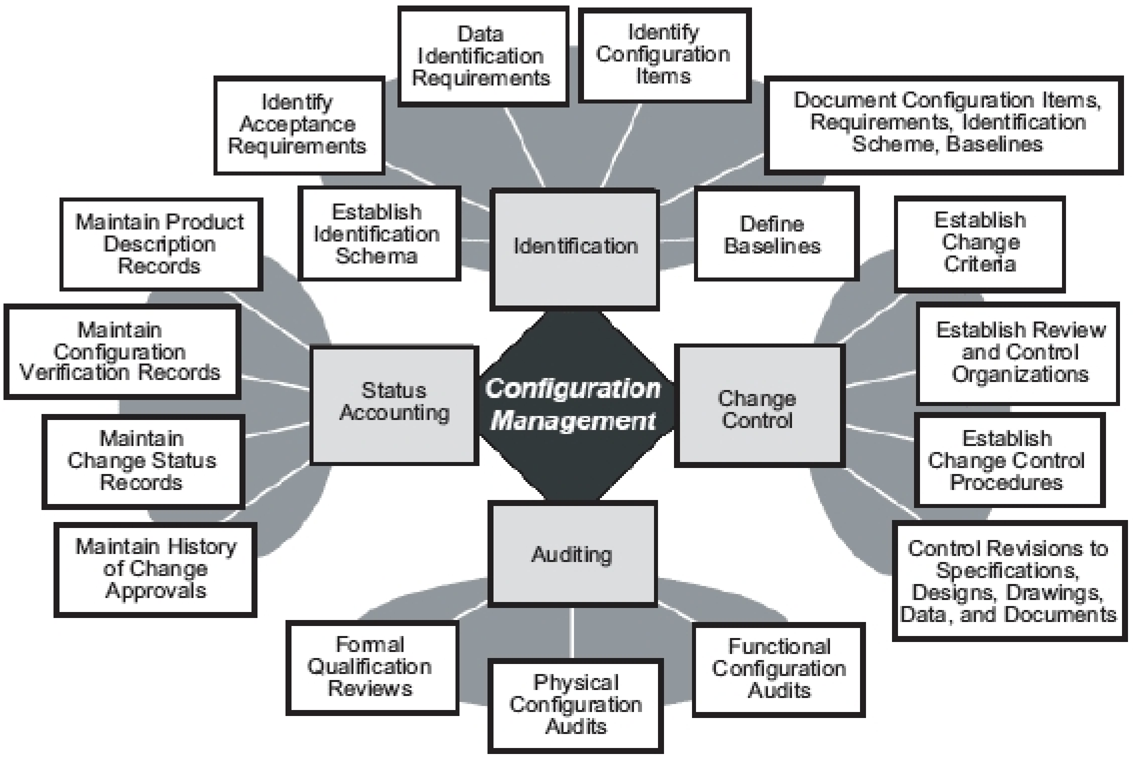

Configuration Management
Identification
Identify Acceptance Requirements
Data Identification Requirements
Identify Configuration Items
Document Configuration Items, Requirements, Identification Scheme, Baselines
Establish Identification Schema
Define Baselines
Change Control
Establish Change Criteria
Establish Review and Control Organizations
Establish Change Control Procedures
Control Revisions to Specifications, Designs, Drawings, Data, and Documents
Status Accounting
Maintain Product Description Records
Maintain Configuration Verification Records
Maintain Change Status Records
Maintain History of Change Approvals
Auditing
Formal Qualification Reviews
Physical Configuration Audits
Functional Configuration Audits

Figure 3.3: Functions of Configuration Management [CENTER 2005]

Identification

The first part of CM is the identification of all items whose configuration needs to be controlled. These configuration items (CIs) can be grouped into hardware, software and documentation (A list of configuration items can be found in table 3.1 Configuration Items of Software Configuration Management). When using a CM suite, which is really the only way to manage a configuration efficiently, identification includes adding all found CIs to the CM system. Most CM suites have their own naming and version number conventions. But even if the application is so small, that the use of a CM tool is not necessary, naming conventions to ease identification have to be applied. Versioning is also started at this level of CM.

After including the items to the CM system, a baseline configuration is established for all important CIs. The idea of a baseline has already been well defined 1978 in [Bersoff et al. 1978]:

> A system baseline is like a snapshot of the aggregate of system components as they exist at a given point in time; updates to this baseline are like frames in a movie strip of the system life cycle. The role of software configuration identification in the SCM process is to provide labels for the contents of these snapshots and the movie strip. [Bersoff et al. 1978]

Any changes of CIs according to the baseline have to be arranged with the configuration manager. The key data of CM and therefore of a baseline are code fragments and requirements, but other components should be included as well. During a development process, after each important stage, a new baseline can be created.

Change Control

Change control has already been introduced at section 3.1.2 Concepts of Configuration Management. The basic functionalities of change control and the change control process have been described. The next step is to apply change control correctly. In order to trace the changes the state of CIs has to be controlled.

> Controlling the release of a product and changes to it throughout the lifecycle by having controls in place that ensure consistent software via the creation of a baseline product. [Dart 1994]

First, it has to be decided which of the identified CIs needed to be controlled. After all, change control is a very time consuming activity and for many CIs it's enough to do only versioning. But all other rather critical resources have to be added to change control. There are different control mechanisms for the different types of CIs. Not every document needs to be watched the same way. All changes to configuration items have to be requested, published and tracked by the CM suite. Developers and Managers are authorized for changing CIs at various levels. In order to get the best control of changes, it's necessary to have change rights so that not everyone can change all CIs.

The control activities of CM include the following [Center 2005]:

- Defining the change process.
- Establishing change control policies and procedures.
- Maintaining baselines.
- Processing changes.
- Developing change report forms.
- Controlling release of the product.

Status Accounting

Now that all configuration items had been identified and also change control had been included to the CM system, the next step is to care about documenting. Status accounting is the part of CM dealing with this topic.

> Status Accounting: recording and reporting the status of components and change requests, and gathering vital statistics about components in the product. [Dart 1994]

Status accounting documents all defined baselines and the established configurations and makes reports about the current configuration status. These reports are also created for the different releases of the application and of the products. Another task of status accounting is to provide a history of all done changes. Additionally, management activities like change requests and authorizations are saved in the CM. To achieve status control at this high degree, a CM suite often needs to maintain a configuration management database (CMDB).

Status accounting has the following fields of activities [Center 2005]:

- Maintain product description records.
- Maintain configuration verification records.
- Maintain change status records.
- Maintain history of change approvals.

Auditing

The last part of CM is auditing. Auditing verifies if the defined configuration equals with the actual configuration. The whole CM process is valueless, if the real configuration does not match with the defined one. Therefore (and as a part of auditing), formal reviews and informal monitoring is done. To make it possible to review configurations, proper status accounting is required.

Auditing is defined as following:

> Audit and review validating the completeness of a product and maintaining consistency among the components by ensuring that the product is a well-defined collection of components. [Dart 1994]

Configuration auditing verifies that the software product is built according to the requirements, standards, or contractual agreement... The goal of a configuration audit is to verify that all software products have been produced, identified and described, and that all change requests have been resolved according to established CM processes and procedures. [Center 2005]

Configuration audit activities include the following [Center 2005]:

- Defining audit schedule and procedures.
- Identifying who will perform the audits.
- Performing audits on established baselines.
- Generating audit reports.

3.1.5 Phases of Configuration Management

Each popular process model calls for realising CM as an activity, but each process model realises CM on a different way. The Rational Unified Process (RUP) for example defines a configuration and change management workflow. This workflow is one of the core workflows of RUP and has to be established in each project. Figure 3.4 illustrates this workflow and the roles involved in it. This workflow is a very specific description of how to do CM in a project based on RUP. For this work, a more general look at the topic of CM is preferred. In section 3.1.4 Parts of Configuration Management, the different components and aspects of CM were described. Based on those parts of CM, a general and simple process for CM is defined. Figure 3.5 illustrates this process for CM.

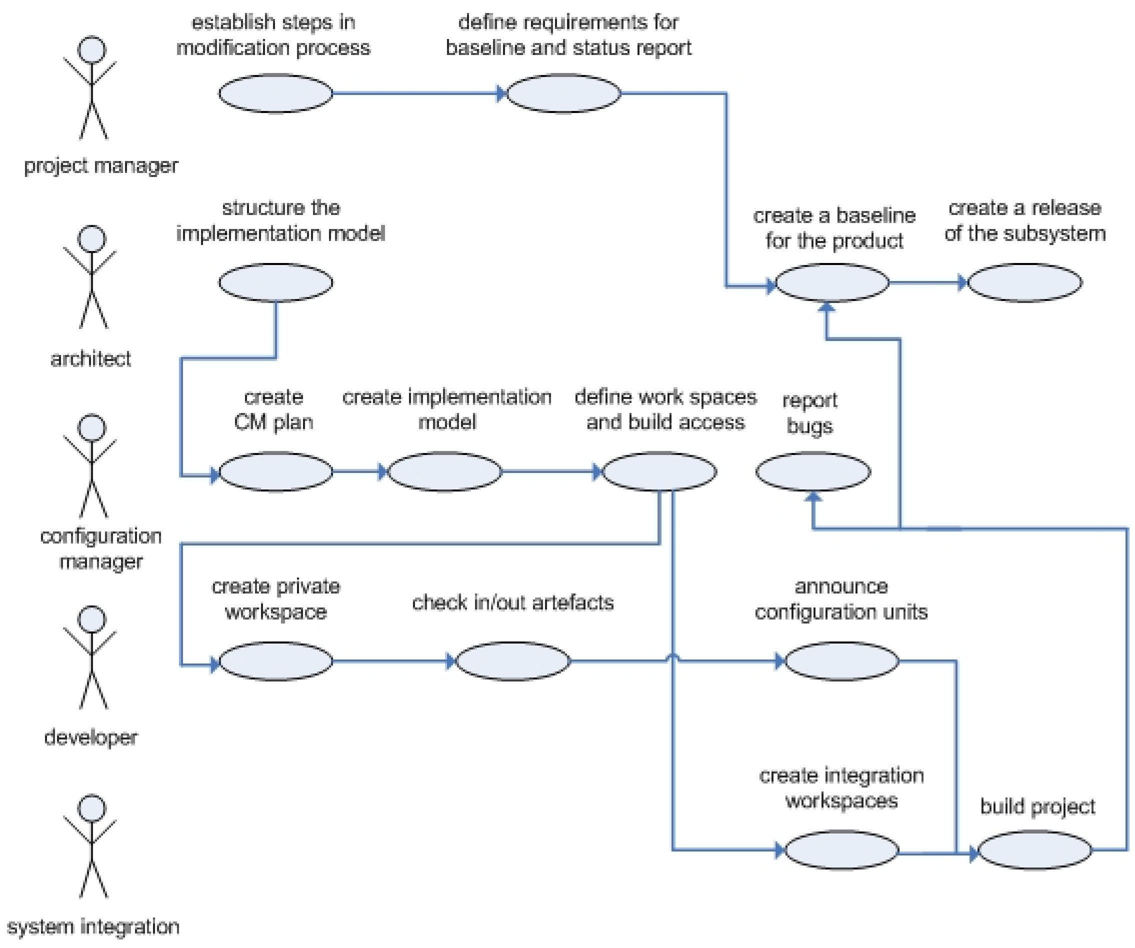

establish steps in modification process
define requirements for baseline and status report
project manager
structure the implementation model
create a baseline for the product
create a release of the subsystem
architect
create CM plan
create implementation model
define work spaces and build access
report bugs
configuration manager
create private workspace
check in/out artefacts
announce configuration units
developer
create integration workspaces
build project
system integration

Figure 3.4: The Rational Unified Process core workflow for Configuration Management

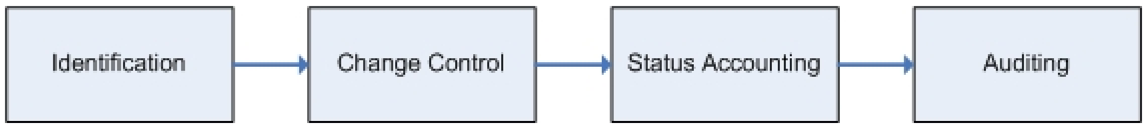


Figure 3.5: Configuration management process

3.2 Model Driven Software Development

MDSD is the key approach for making MDCM possible. Most concepts of MDCM are based on it. Unlike traditional software development, MDSD uses models not only for documentation purpose, but also as a foundation for implementation and development. By using transformations, information is added to models and code is generated. MDSD doesn't dictate any technologies, it only offers an approach.

Nevertheless, in order to successfully apply MDSD, three requirements must be met [STAHL and VOELTER 2006]:

- Domain-specific languages are required to allow the actual formulating of models.
- Languages that can express the necessary model-to-code transformations are needed.
- Compilers, generators or transformers are required that can run the transformations to generate code executable on available platforms.

3.2.1 Model Driven Architecture

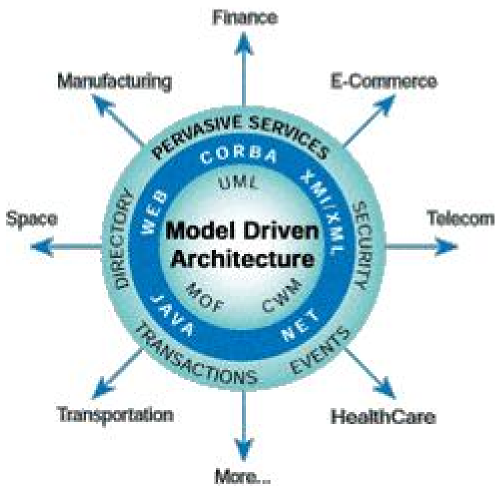


Figure 3.6: OMG's Model Driven Architecture
(Overview of the different components and application areas of the Model Driven Architecture approach from the Object Management Group)

Model Driven Architecture (MDA) is a strategy and industry standard of OMG (Object Management Group).

> MDA: An approach to IT system specification that separates the specification of functionality from the specification of the implementation of that functionality on a specific technology platform. [PETRASCH and MEIMBERG 2006]

MDA is very similar to the MDSD approach. But unlike MDSD, MDA is focusing on UML-based and other OMG associated modeling languages.

> The primary goal of MDA is interoperability between tools and the long-term standardization of models for popular application domains. In contrast, MDSD aims at the provision of modules for software development processes that are applicable in practice, and which can be used in the context of model-driven approaches, independently of the selected tool or the OMG MDA standard's maturity. [STAHL and VOELTER 2006]

The book [ZEPPENFELD and WOLTERS 2006] describes the concepts of MDA including the models of UML. It also shows the appliance of MDA on a small project. An overview of current research in MDA/MDSD and of the major challenges that must be tackled can be found in the paper [FRANCE and RUMPE 2007].

3.2.2 Procedure models and workflows

MDCM tries to combine MDSD with CM. MDSD has to be classified somewhere between process/procedure models and development methods. In order to be able to use it for a professional project, a procedure model is required. A procedure model simplifies the whole development process and provides a foundation for best practice workflows. A good way to integrate MDSD into a common software development process is to use a procedure model and enhance it with its aspects. Figure 3.7 shows a V-model XT including MDSD methods.

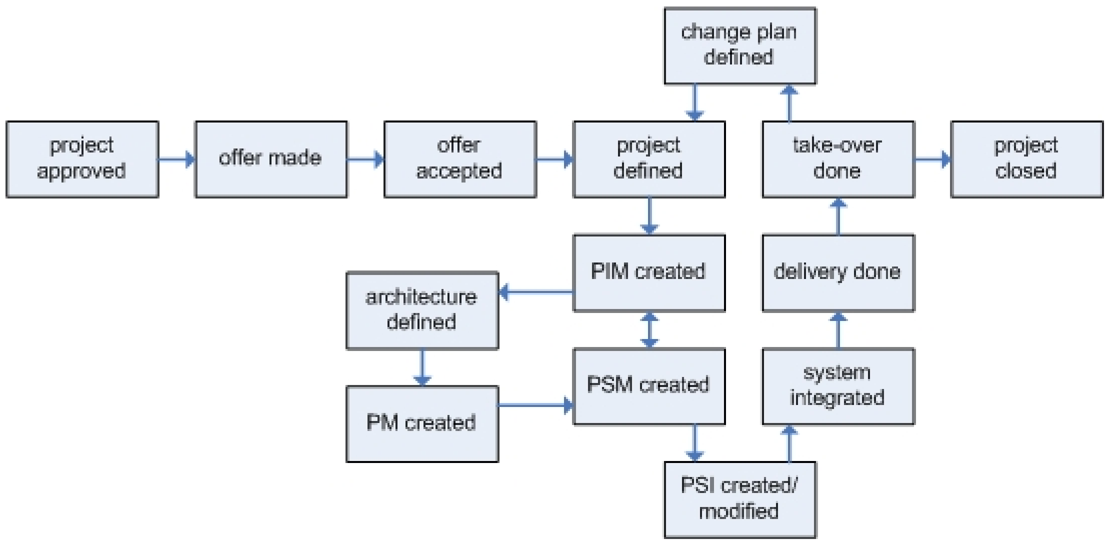


Figure 3.7: V-model combined with the Model Driven Software Development approach (based on [PETRASCH and MEIMBERG 2006])

Another way to solve the integration problem of MDSD is to use its paradigm as a sum of artefacts, phases and tasks performed as a part of a standard process model. The next sections introduce the basic artefacts and phases of MDSD, which will also be used in MDCM.

3.2.3 Models of Model Driven Software Development

Although there is no specific procedure model for MDSD, there is an approach how to use it. This approach describes the artefacts created at each phase of the development process. Before each phase of MDSD will be explained, at first the basic artefacts/models of it are described. Figure 3.8 illustrates how the different model are connected.

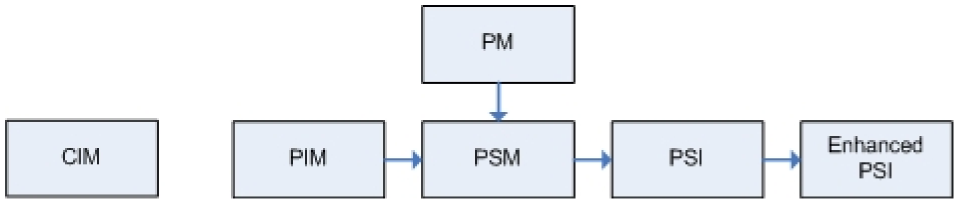


Figure 3.8: Model hierarchy of Model Driven Software Development

Computation Independent Model (CIM)

The CIM (also known as domain model) describes the system without concerning hard- or software. It explains the system and its functions without the need to meet a concrete syntax. CIM shows the business model of the system and should be easy to understand by everyone involved in the project.

Platform Independent Model (PIM)

The PIM describes the system without specifying a platform for it. The PIM has to meet the formal requirements of MDSD and provides the foundation for the whole transformation process.

Platform Model (PM)

The PM defines a specific platform for a software system. The platform specific model is created through a model transformation with PIM and PM as input.

Platform Specific Model (PSM)

The PSM describes the system for a specific platform. The PSM has already very detailed information about both, the target platform and the future application.

Platform Specific Implementation (PSI)

The PSI, which is already executable code, is generated using the PSM. It's the implementation of the system which has to be manually extended now.

3.2.4 Phases of Model Driven Software Development

Each of those models belongs to a specific phase of the MDSD development approach. Figure 3.9 shows the different phases and their iterations. Like in IBM's rational unified process it is also recommended to use iterations for MDSD.

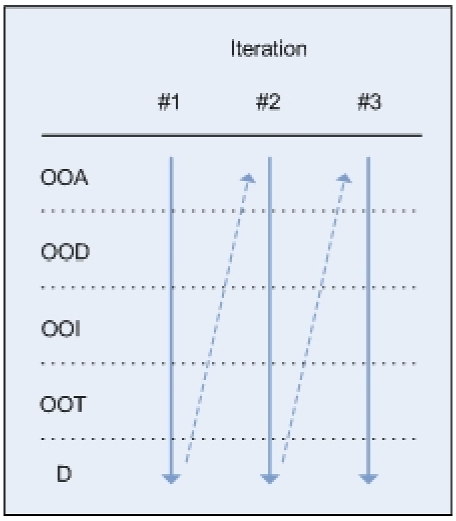


Figure 3.9: Iteration of the Model Driven Software Development phases

Object Oriented Analysis (OOA)

Like system analysis, OOA creates a concept of an application. It is concerned with developing software engineering requirements and specifications which are expressed as object models. As a result of OOA, CIM and PIM has been created.

Object Oriented Design (OOD)

OOD or software design extends the concept of OOA and adds platform specific and more detailed information to it. OOD is concerned with developing an object-oriented model of a software system to implement the identified requirements. Artefacts of this phase are the PM, the PSM which is generated using the PIM and PM and the PSI based on the PSM.

Object Oriented Implementation (OOI)

The generated PSI code as foundation, the implementation is manually extended. Since not every aspect can be modeled, still a lot of development is done in this phase of the MDSD project.

Object Oriented Testing (OOT)

Important parts of the implementation and the whole application are tested, errors and problems are solved.

Deployment (D)

The tested and finished application will be at last deployed so that the customer is able to use it.

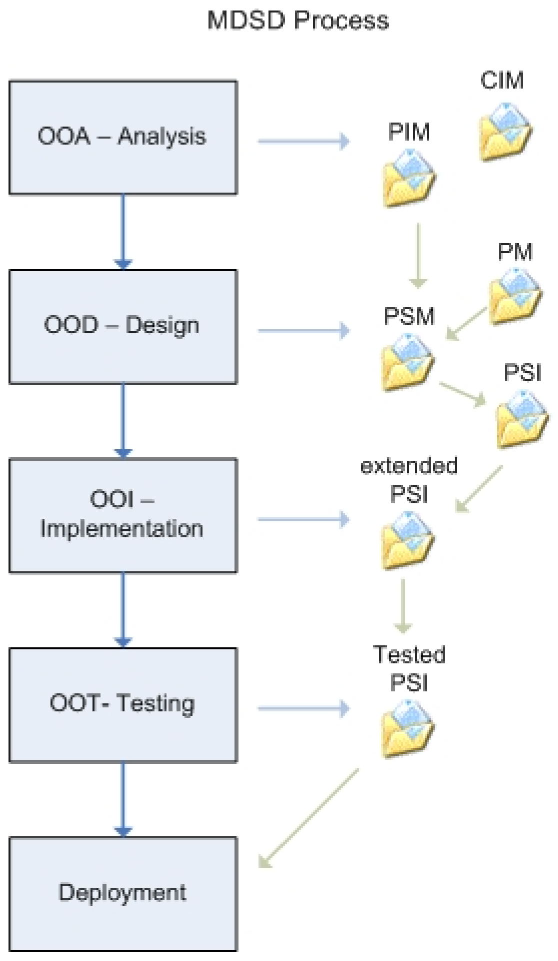


Figure 3.10: Model Driven Software Development process

Figure 3.10 illustrates the phases of MDSD and their artefacts. This model is a kind of best practise recommendation for how to do MDSD.

3.3 Unified Modeling Language and Eclipse frameworks

Modeling languages, especially EMF, are the key technologies enabling MDCM. The invention and rise of modeling languages, UML in particular, changed the whole software engineering. This section describes the two important languages for MDCM and the most important feature of both, metamodeling. The next two sections should only give a brief overview of the modeling technologies [2]. The third section covers GMF, which uses EMF models as a foundation to create graphical editors on top of it.

3.3.1 Unified Modeling Language - a short overview

The Unified Modeling Language (UML) is a standard of the Object Management Group (OMG) to model software and other systems. The goal of OMG is to provide a standardized modeling language and a graphical notation for static and dynamic structures. UML is the dominating language for modeling software systems. Everyone should be able to read and understand the different UML diagrams, since they are often used to support communication between different roles of a software company. The different diagrams of UML are divided into the following parts:

- **Structure diagrams** describe the components of a system and provide a static view. The class diagram for example is a structured diagram.
- **Behavior diagrams** model the activities of the system. Activity and Use case diagrams are a part of this group.
- **Interaction diagrams** like the Sequence diagram describe the flow of control and data between the different components of a system.

The role of UML in Model Driven Architecture can be found in the book [FRANKEL 2003].
Metamodeling and the Meta Object Facility

The Meta Object Facility (MOF) is the language used for metamodeling in UML. A Metamodel is a model that describes other models. This definition of Michael Blaha provides a good overview of what metamodeling is. To cover this topic adequate, a more specific description is needed.

[2]More specific information can be at `http://www.eclipse.org/modeling/emf/` and `http://www.uml.org/`

Modeling describes a specific problem or task using a model which illustrates the relevant parts of a real world system. A model is defined by a modeling language. The modeling language is defined by a metamodel. The modeling language used for the metamodel is the same as used modeling a model itself. The benefit of this approach is, that a model and a metamodel have the same structure and can be read the same way. As a model describes the components and connections of the system, the metamodel describes the elements of the model. One layer higher, the metamodel is again defined by a metametamodel. The following diagram shows the relation between the different layers.

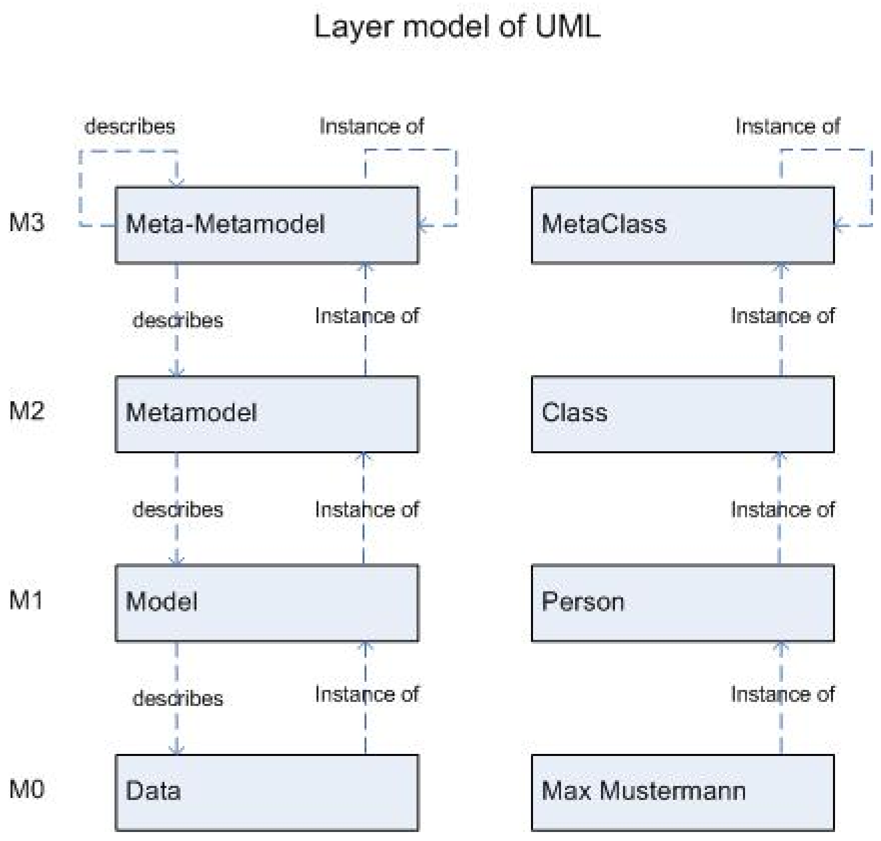


Figure 3.11: Layers of the UML model including an example

The left side of the figure shows the layers M0 to M3 and their names. On the right side, there is an example, which describes the layers using a class person.

Metamodeling enables a completely different new approach for software engineering. MDA as well as MDSD are approaches based on the invention of metamodels.

The following quote illustrates the importance of metamodels:

Metamodeling is the base for domain specific or architecture centered mod-

> eling, tool adaptation, model validation, model transformation and code generation.[VOELTER 2003]

MDCM follows the concept of using UML diagrams to provide the foundation for CM. The paper [KOEGEL 2008] on the other hand extends software configuration management to support better change management on UML diagrams. Overall it's obvious that there however it is done there is a big need to include UML into CM.

More information on how to use UML can be found in the book [JECKLE et al. 2003].

3.3.2 The Eclipse Modeling Framework

EMF is an Eclipse Plug-in comparable to UML. However, UML has a greater complexity. The following definition is from the project site of EMF.

> EMF is a modeling framework and code generation facility for building tools and other applications based on a structured data model [3].

EMF provides tools to produce a set of Java classes and an editor for the model. The format of an EMF model is XML Metadata Interchange (XMI). These models can be defined using annotated Java, UML, XML documents or the EMF editor. Addtionaly, UML diagrams of many tools like Magicdraw or RSA can be transformed into an EMF model. EMF also provides the foundation for many tools and applications (for example GEF or GMF). Ecore is the part of EMF, that is used for defining metamodels. The book [QUATRANI and PALISTRANT 2006] is about how IBM is using these technologies for visual modelling. The following figure illustrates an EMF editor.

[3]from `http://www.eclipse.org/modeling/emf/`

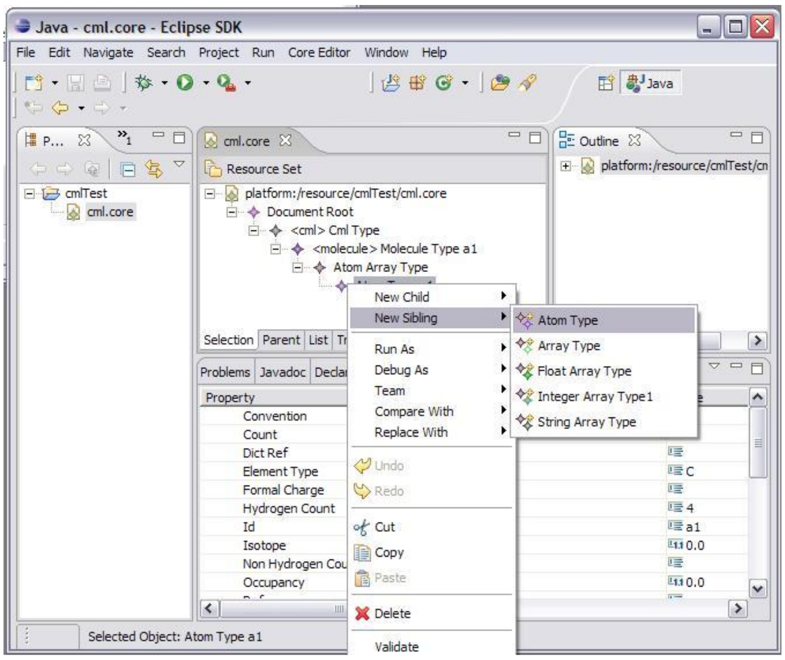


Figure 3.12: The Eclipse Modeling Framework editor

3.3.3 The Graphical Modeling Framework

MDCE, the graphical editor for XML configurations and CM models, is an important and big part of MDCM. The MDCE is using EMF, which has been introduced in section 3.3.2 The Eclipse Modeling Framework, as modeling language and GMF for the graphical representation. The goal of this section is to give an overview of GMF and to provide enough information to understand how the MDCE works [4].

The Graphical Modeling Framework (GMF) provides a generative bridge between Eclipse Modeling Framework (EMF) and Graphical Editing Framework (GEF). The GEF allows developers to create a graphical editor from an existing application model. GEF uses the org.eclipse.draw2d for displaying graphics and the MVC (Model View Controller) architecture to separate the model, view and logic. GEF can be used to build nearly every diagram type and it is very powerful but also complex. GMF instead has its focus on developing modeling editors. The functionality of GMF is more limited compared to GEF, but therefore it is also less complex and better qualified for our intention.

The GMF uses five different components/models to define the behaviour and look of the editor and its relation to the underlying data definition. The next diagram [5] illustrates the main

[4]Further information regarding GMF can be found at `http://www.eclipse.org/gmf/`
[5]from `http://wiki.eclipse.org/index.php/GMF_Tutorial`

components and models used during GMF-based development.

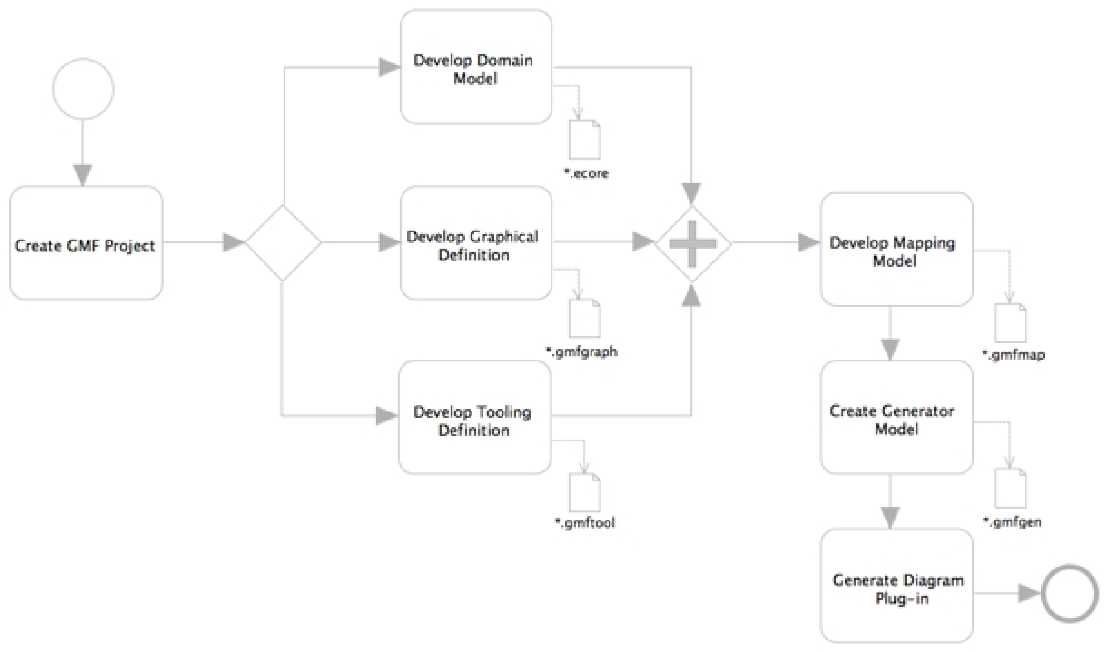


Figure 3.13: Graphical Modeling Framework Overview

Domain Model Definition

The Domain Model Definition is a metamodel defined in Ecore or using XSD. The EMF components, needed by GMF are generated out of it. So the Domain Model Definition is not really a part of GMF, but is prerequisite for starting.

Graphical Definition

The Graphical Definition contains all graphic elements used by the editor. The presentation of the objects and connections is defined here. The Graphical Definition is the part of GMF responsible for the whole looks. The paper [EHRIG et al. 2005] describes the basic steps to create visual editors using EMF and GMF. The most important elements of GMF are listed below.

Basic components

- The **Figure Gallery** defines the presentation of the different elements of the model. Rectangles, labels, connection lines with arrows or without, the layout and colors are defined here.
- Every model element has it's associated **Node** item. Every Node is presented by a Figure Gallery element. A Class of the model will be mapped into a node.

- The **Connection** represents a relation between nodes and is also linked to a Figure Gallery item.
- The **Diagram Label** is linked with a Figure Gallery label. Labels, defined at this level, are later connected with the attributes of the model classes.

Nodes, Connections and Diagram Labels are the bridge from the presentation to the model. Nodes are related to Classes, Connections to Relations and Labels to Attributes. The relation will be specified in the Mapping Definition, but the elements have to be defined here.

The Figure Gallery

The Figure Gallery is the core of the Graphical Definition. The whole look is defined here. The following list gives an overview of the different presentation elements.

- **Rectangles**, **Ellipses** and **Polygons** are the basic elements. Those elements can be nested into each other.
- **Connections** and **Decorations** can be used to show relations between those elements.
- **Labels** display text.
- **Layout**, **Size** and **Location** define the position and arrangement of all elements.
- With the help of **Custom Figures**, **Custom Connection** and **Color Definitions**, an individual look can be achieved.

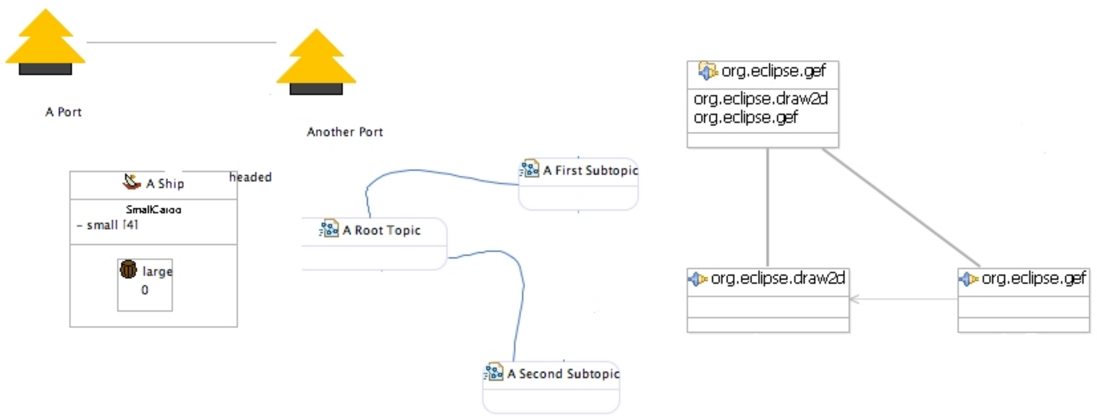


Figure 3.14: An Overview of GMF elements
(Shows some of the graphical elements of GMF. These elements can also be changed or enhanced.)

Tooling Definition

The Tooling Definition contains the menu bar and the Icons of GMF.

The Tooling Definition model is used to specify the palette, creation tools and actions for the graphical elements. A good description of the Tooling Definition is:

> Currently, the Tooling Definition is basically used to describe desired diagram palette tools set, which consists of standard tools like Selection Tool, Zoom Tool and Creation Tool. It is possible to organize tools into groups and attach appropriate icons. [SHATALIN and TIKHOMIROV 2006]

Although there are other available control elements of the Tooling Definition model, like a Pop-up menu or a main menu, these are not considered yet. The palette offers enough functionality for MDCM.

Mapping Definition

The Mapping Definition merges the preceding definitions and defines the relations between the items. The graphical object of the Graphical Definition, the data objects of the Domain Model Definition and the menu items are connected. The generator model is created out of the mapping definition. The Mapping Definition is the most important model of GMF, because it fits all together. Therefore, the major elements are explained more precisely.

- The **Node References** define the elements of the editor. Top Node References represent items which will be displayed on the top drawing area. Other Node References are children of the Top Node References and will therefore be painted inside the Top Node Reference element. A Node Reference has to be related to an EClass element of the data model, to a Node element of the Graphical Definition and to a Creation Tool of the Tooling

Definition.

- The **Label Mappings** are text fields for the input of the attribute values. They are linked to Diagram Labels and to the EAttributes of the data model. Label Mappings are children of Node References.
- The **Link Mappings** represent the connections of the different Node References. They are related to an EReference element and an EClass of the data model. Source and target of the link have to be defined. Additionally, a Creation Tool is related to the Link Mapping.

Generator Model

The GMF editor is built with the generator model. Some minor modifications can be done in the generator model like changing the name of the generated editor and so on.

3.4 Transformations

Transformation is one of the most important parts of MDSD as well as of MDCM. The idea of both is the same, to use transformations to generate different kinds of new components based upon a single starting point. In MDCM, transformations are used to build models from XML Schemas and to transform models to XML configurations or CM input. A theoretical view on using XML processing for transformations in an model driven approach is described in [IVAN KURTEV 2003].

There are three technologies, which will be used for transformation:

- XSD to model transformation

  To provide a comfortable entry to MDCM, this work introduces some possibilities to use XSDs to generate UML and EMF models. There are some tools available which offer such transformations. Additionally, this work shows some rules to manually map an XSD to a model.
- Mapping with openArchitectureWare (oAW)

  OpenArchitectureWare is an open source MDSD framework. There are model-to-model and model-to-code transformations available. With the model-to-code transformation, a model (in our case the created configuration) can be transformed into any code (here the XML configuration file). OAW offers a template language and a workflow for execution. The transformation is based on the Ecore metamodel. Writing the transformation is straightforward. However, running the transformation as an Eclipse Plug-in is very difficult.
- XML transformations with JDOM

  Even JDOM is not an MDA or MDSD tool, it can be used here for transformation purpose. JDOM is a Java library for reading, writing and changing XML files. JDOM uses the

Document Object Model (DOM) and offers methods for building, changing and saving DOM trees. Since the editor created file is written in XML as is the target file, JDOM can be used for an XML-to-XML transformation. The implementation effort is a bit higher than using oAW transformations, but it's easier to add additional queries and features to the transformation process. Additionally, it's no problem to run a JDOM transformation as a Plug-in.

3.4.1 openArchitectureWare

OpenArchitectureWare (oAW) is a Model Driven Software Development (MDSD) framework implemented in Java. It offers model-to-model and model-to-code transformation. OAW provides tools and functions for MDSD and supports many import and output formats. It uses models of EMF, of UML tools like MagicDraw or the Enterprise Architect and models of Eclipse UML2. The output of oAW can be Java or C/C++ code, an XML file or some documentation. The tooling of oAW like editors and model browsers is based on the Eclipse platform. OAW also includes many components of other model driven development related projects (for example, oAW provides a GMF integration). OAW has a modular design where nearly any of the framework components can be enhanced and extended. It is used in all kinds of different domains like .Net applications, Eclipse based Java projects and J2EE development. The following image illustrates the different components of oAW, described in the next section.

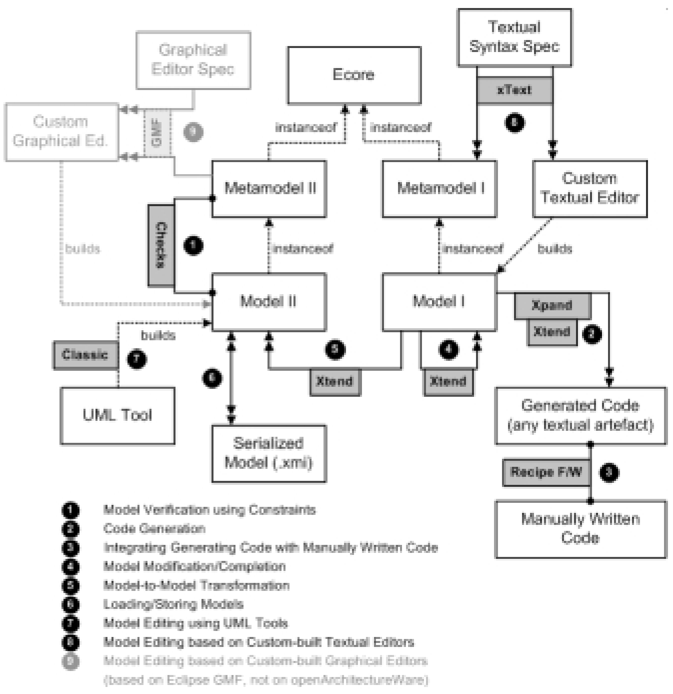

Graphical Editor Spec
Ecore
Textual Syntax Spec
xText
Custom Graphical Ed.
GMF
instanceof
instanceof
Metamodel II
Metamodel I
Custom Textual Editor
builds
Checks
instanceof
instanceof
builds
Model II
Model I
Xpand
Xtend
builds
Classic
Xtend
Xtend
UML Tool
Serialized Model (.xmi)
Generated Code (any textual artefact)
Recipe F/W
Manually Written Code
1 Model Verification using Constraints
2 Code Generation
3 Integrating Generating Code with Manually Written Code
4 Model Modification/Completion
5 Model-to-Model Transformation
6 Loading/Storing Models
7 Model Editing using UML Tools
8 Model Editing based on Custom-built Textual Editors
9 Model Editing based on Custom-built Graphical Editors (based on Eclipse GMF, not on openArchitectureWare)

Figure 3.15: OpenArchitectureWare core features overview from [VOELTER 2007]

Core features

The core of oAW is the workflow engine. Every activity in oAW has to be started at the workflow engine. There are a number of prebuilt workflow components for the basic functionality. The core language of oAW comprises of the Xpand, Extend and Check language. All three are based on the same expression language and type system called Expression Engine. As a next step, the core components are described.

Workflow engine

> The oAW4 workflow engine is a declarative configurable generator engine. It provides a simple, XML based configuration language with which all kinds of generator workflows can be described. [EFFTINGE and VOELTER]

Typically, a workflow calls one workflow component after another. A workflow component can be a model parser, model validator, model transformer or code generator. An often used basic component sequence is to start the workflow calling a model parser component to get the model input, then validate this input and call the code generator to create code out of the model. There are already many oAW cartridges available, which can be added to and started by a workflow. A cartridge provides a concrete function and extends the behaviour of the workflow. For example, there is a cartridge for generating hibernate files out of a model.

Xpand language

Xpand is a template language used by oAW to generate output. An Xpand template file is started by the workflow and receives a model as input. This model can then be mapped into an other model or into code. Step by step Xpand handles each model element and attribute to generate the output. Xpand is the heart of an oAW transformation process.

Extend language

The Extend language offers additional possibilities to the Xpand language and oAW framework. With Extend, methods can be written to enhance the transformation process. There is an own Extend language based on the oAW Expression language. Alternatively, a Java method can be written and executed using the Extend language, which is very convenient for Java developers. An Extend language method can be invoked using the workflow engine or at an Xpand template.

Check language

> An important concept in model-driven software development is the idea of having domain-specific languages (DSLs) that are used to describe domain-specific stuff in a short and concise manner. Check is a domain-specific language that is specialized in model validation. [EFFTINGE]

Not only oAW offers a Check language based on the Expression language for validating models, it also provides OCL integration. OCL is a part of UML and specificies invariants of class diagrams, preconditions/postconditions for methods and many more conditions.

3.4.2 JDOM - Java Document Object Model

JDOM is an API for reading, manipulating and writing XML documents. It is an alternative to the DOM and SAX API, even it integrates well with both. The following list shows advantages of JDOM to the standard DOM or SAX.

- JDOM was designed specifically for Java rather than for multiple languages which guarantees consistent programming.
- JDOM offers the same functionality, is easier to use, has a better performance and less memory requirements

1.0 is the current version of JDOM. JDOM is organized as a JAR archive which has to be added to the standard Java library to use it. Regarding the documentation: There are many articles about JDOM published at the project homepage and a JavaDoc is available.

The components of JDOM

Two very important classes are the org.jdom.input and output classes. They provide the methods for building a JDOM document out of an XML or DOM document and for transforming the JDOM document back to XML. The following picture [6] illustrates this.

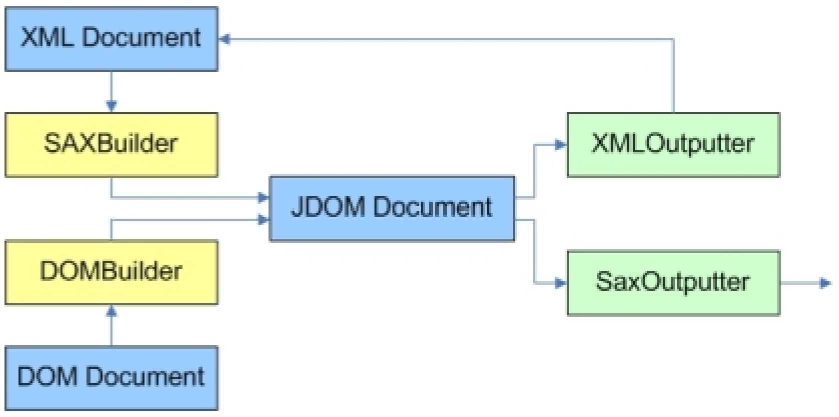


Figure 3.16: JDOM - input and output

(XML documents can either be parsed by SAX or DOM parser to create a JDOM document. This document can be manipulated or used to create again a SAX document or an XML document. A JDOM object can also be created from the scratch.)

[6]from http://www.jdom.org/

The most important classes of JDOM except the already mentioned input and output classes are the following:

- The **Document class** represents the XML-Document. It has to have a root element where the processing and scanning of the document can start.
- The **Element class** represents an XML-Tag. An element can have children, text and attributes.
- The **Attribute class** is used to create attributes for the elements.

3.4.3 Creating a model out of an XML Schema

Since the invention of XML Schema in 2001, XML Schema Definitions are the common way to define the structure of XML files. There are many tools for XSD files, which display them in a tree based manner and therefore offer an easier editing and a better illustration. Due to the fact, that UML is also very popular, there are also many tools available for transforming an XSD to a UML Class Diagram. Such a transformation is very important for MDCM, because it provides an easy entry to the whole topic of modeling configurations. Usually configurations are written in XML and their structure is based on an XSD. By transforming this schema a model is created which already represents the structure of the configuration and which can be used as foundation for the further MDCM workflow steps. There are many possible approaches on how to convert an XML or XML Schema into a model, one interesting paper covering this topic is [Hou et al. 2001].

The most popular tools enabling a good clean conversion are:

- Rational Software Modeler
- MagicDraw
- XMLSpy

Freeware products which offer a transformation are rather rare. Most of the freeware applications offer either XML or UML editing, but hardly both. The Eclipse Plug-in HyperModel is specialized in creating and converting XML Schema Definitions. It is discussed in this section, on page 58. The conversion of an XSD to an EMF model is offered by EMF itself. Based on an XSD, an Ecore Model can be generated automatically. A model editor based on the XSD for the XML files can then be generated and used. More information on this topic can be found on page 57.

The conversation from XSD to model comes very handy when using MDCM because most of the configurations based on XML also have a XML Schema defining them and therefore it is possible to generate the needed models automatically.

Rules for mapping an XSD to a model

In order to convert XML Schema Definitions to a model, rules are necessary. The following basic rules are a short extract of the work UML Documentation Support for XML Schema. [Salim et al. 2004]

XML Schema Constructs	UML Constructs
Global element	Class
Local element	Attribute
Global attribute	Class
Local attribute	Attribute
Optional constraint	Multiplicity [0..1]
Required constraint	Multiplicity [1..1]

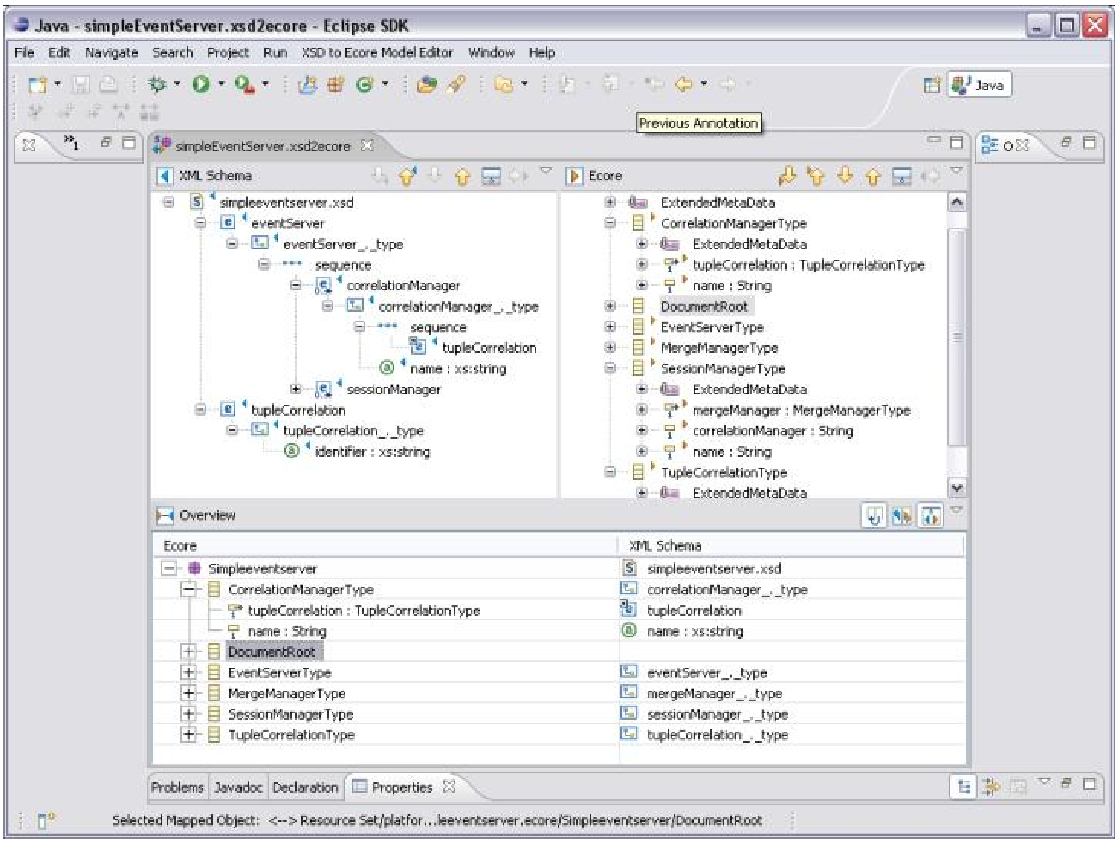
Table 3.2: Mapping Schema Types to UML

Another paper dealing with the conversion of XML Schema Definitions to models and with the design of XML Schemas using UML diagrams is called UML and XML Schema [Routledge et al. 2001]. It separates a model into three common layers to solve the mapping problem more accurate. The three layers are called the conceptual, logical and physical layer. This is a classic three layer architecture used by many modeling approaches in order to be able to concentrate on each topic and not mix them up. This is an unusual but very promising method of setting UML and XSD documents in relations.

Converting an XSD to Ecore

Converting an XSD to an Ecore model is easy to achieve. In Eclipse, when creating a new EMF model, it is possible to choose the model which the EMF model will be based on. This model can be an XSD, a UML model and more. After selecting the XSD, an Ecore model, the EMF genmodel and optional an Ecore-XSD-Mapping are created. Tags and their children tags are modeled as classes and relations, Schema attributes are converted to EMF attributes and so on. There are few minor issues on the conversion result like the cardinality of the EMF relations is not always correct. But overall, the conversion works fine and if not everything is mapped right, there is still the possibility to change the transformation by editing the XSD2Ecore file. After all, the transformation works the better the more specific the XSD is defined (for example, when the occurrence constraints are defined at the XSD, the cardinality is mapped correctly).

Figure 3.17 shows the XSD2Ecore file which illustrates how the XSD elements are transformed.


Java - simpleEventServer.xsd2ecore - Eclipse SDK
File Edit Navigate Search Project Run XSD to Ecore Model Editor Window Help
Java
Previous Annotation
simpleEventServer.xsd2ecore
XML Schema
Ecore
Overview
Problems Javadoc Declaration Properties

```
XML Schema
simpleeventserver.xsd
  eventServer
    eventServer_._type
      sequence
        correlationManager
          correlationManager_._type
            sequence
              tupleCorrelation
            name : xs:string
        sessionManager
  tupleCorrelation
    tupleCorrelation_._type
      identifier : xs:string
```

```
Ecore
ExtendedMetaData
CorrelationManagerType
  ExtendedMetaData
  tupleCorrelation : TupleCorrelationType
  name : String
DocumentRoot
EventServerType
MergeManagerType
SessionManagerType
  ExtendedMetaData
  mergeManager : MergeManagerType
  correlationManager : String
  name : String
TupleCorrelationType
  ExtendedMetaData
```

Ecore	XML Schema
Simpleeventserver	simpleeventserver.xsd
CorrelationManagerType	correlationManager_._type
tupleCorrelation : TupleCorrelationType	tupleCorrelation
name : String	name : xs:string
DocumentRoot	
EventServerType	eventServer_._type
MergeManagerType	mergeManager_._type
SessionManagerType	sessionManager_._type
TupleCorrelationType	tupleCorrelation_._type

Selected Mapped Object: <--> Resource Set/platfor...leeventserver.ecore/Simpleeventserver/DocumentRoot

Figure 3.17: The XSD2Ecore Model in Eclipse
(The XML Schema on the left side and the matching Ecore model on the right side. The editor is a part of the Eclipse EMF Plug-In.)

There is also a very good Wiki document about generating a dynamic Ecore model out of an XSD. It shows the Java code which is necessary to do this on runtime and also shows how to load an XML document into the generated model [7].

Converting an XSD to a UML Model

Many UML tools and even XML tools offer a conversion between these two formats. Next, this work provides a closer look on MagicDraw and on HyperModel, an Eclipse Plug-in.

[7]See http://wiki.eclipse.org/

HyperModel

HyperModel [8] is a free Eclipse Plug-in for defining XML files, XML Schema definitions and convert them into UML class diagrams. XSD and XML files can be easily imported and integrated into a HyperModel project. HyperModel offers an editor for XML files, XML Schemas and UML Class diagrams. HyperModel is integrated into Eclipse and is there for easy to use for everyone familiar with the Eclipse workbench. HyperModel has a wizard for converting XSD files to UML models. After conversion, the model can be modified and displayed. This all works fine and is handled intuitively. Unfortunately, sometimes the conversion doesn't work well. The model file is then created, but cannot be displayed by the UML editor of HyperModel. Another big disadvantage is the file format of the UML. It is not compatible to any other UML file format and can therefore just be used for presentation purpose. Here is a picture of the generated UML diagram of an XSD for the Naiad Project (which will be introduced in chapter 7.2 The flightplan example.

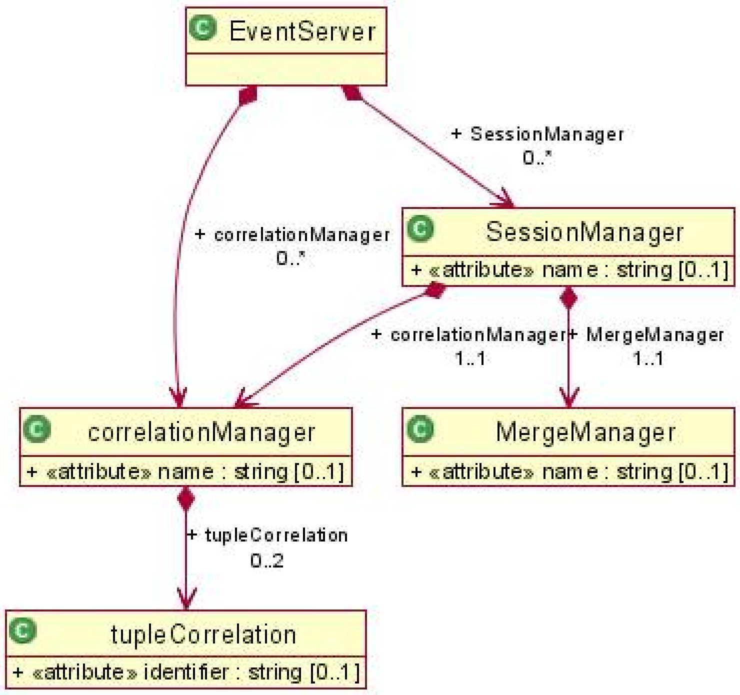


Figure 3.18: A UML diagram using the Eclipse Plug-In HyperModel

MagicDraw

MagicDraw is a UML tool which has a great number of functions, templates and wizards to ease the work with it. But there is so much functionality, that it can be very confusing to work with MagicDraw. Nevertheless, it is able to write UML models as well as XML Schema definitions and convert them into different other models. There are many settings which can be modified when transforming a UML model into an XSD. The transformation process can be adjusted in nearly every way, but there is one big disadvantage when using MagicDraw. It is not possible to import an existing XSD into the workbench. The XSD has to be defined in MagicDraw in order to be able to work with it.

[8] `http://www.xmlmodeling.com/hyperModel/`

4 Model Driven Configuration Management - Feasibility Study

This chapter describes a prototype of a basic Model Driven Configuration Management (MDCM) realisation. It is only a very general description, which should provide a better understanding how a concrete implementation of MDCM might look like.

4.1 The idea behind Model Driven Configuration Management

So far, configuration management has many definitions, is described by many process models and supported by many tools. But there is no standardized language for defining either configuration mangement (CM) processes or the modeled configuration items (CIs) of software configuration management (SCM). The model driven approach gains acceptance in nearly every software discipline. Why not using the model driven aspects for configuration management including change control, release and requirements configuration?

When thinking of defining a software configuration, most people already abstract the given software into models. There are development tools, a database suite, a workspace and the application with all its classes and packages. In order to manage those configuration items, only the needed information has to be extracted out of the real world software components, which usually leads to collect these informations in models. By looking at those items in term of models, they also should be defined as models. The paper [Render and Campbell 1991] describes how to represent SCM items as objects and classes. But unlike MDCM, it has it has a strong focus on version control and also doesn't include a model driven approach.

As an example, when managing an Eclipse installation, most times only a few important properties of the Eclipse installation must be considered. And these properties are:

- Eclipse version
- Java version
- Additional Java libraries
- Plug-Ins
- The workspace directory

By defining only these properties, enough information is provided to manage most Eclipse installations. This abstraction is possible and necessary for all configuration items. The advantages of defining all configuration management aspects as models are obvious. Instead of dealing with different implementations for describing configuration items and their relations for each tool, there would be a clear defined and well known language to define configurations and requirements. The first step to establish Model Driven Configuration Management is to define metamodels for describing the configurations. After defining a language for MDCM, the next step would be to modify tools to support Model Driven Configuration Management. At best it could be possible to define a model of the application, which will be transformed into Java classes, the database structure and into configuration items added automatically to the CM suite. This approach would be a merge of Model Driven Software Development (MDSD) and configuration management. In this scenario, only a few models providing all information necessary for the whole development process. All relations between CIs could be described by models, which also might be used to generate application code or database tables as a part of Model Driven Architecture (MDA). The basic structure of UML is already suited to support CM modeling and UML is a well known language. Most companies already work with UML and there is no additional effort to introduce UML as an approach to define CIs. With support of CM suites it might soon be possible to be able to write models for the whole CM process.

4.2 Defining configurations

> The major process frameworks (ITIL, COBIT, CMM) all call for it. According to ITIL, Configuration Management's goals are to account for all the IT assets and configurations within the organisation and its services. What's in scope? All applications, components, databases, servers, message queues, ETL jobs, and so forth become CIs, or Configuration Items. [Capadouca 2004]

There are different types of items which have to be managed. These items have to be classified, since they require different modeling approaches. In general, there are the following categories of Configuration Items (CI):

- **Hardware components** representing all applied hardware and the whole network environment. From servers to routers to network connections, each hardware important for CM belongs to this group. Since the focus of MDCM lies on software configuration management, hardware components are not covered in greater detail.
- **Software components** consist of all different applications, like the operating system, the development suite, databases and more.
- **Development**: CIs of this group are directly involved in the development process. These CIs include documentation as well as code files. The main focus of MDCM lies on this group of items.
- **Organisation components**: In order to manage access rights or user specific change control, it might as well be also necessary to define organisational units.

Not only that these CI groups have to be managed, also the different management disciplines, which are a part or rather an enhancement of CM, namely requirements management, change control and release management, have to be considered.

But all different CI groups and management disciplines can use the same model driven approaches to be described next.

Abstraction of content

A very basic approach of Model Driven Architecture is to abstract the properties of the modeled items. Only those characteristics necessary for the model are processed and saved. By using this mechanism for defining CIs, only the needed aspects are considered.

Metamodeling

By providing metamodels for the different groups of CIs, the development of the models for CIs will be easier. It is possible to define the attributes of each CI type, the relations and restrictions. By using metamodels as base for defining CM models, a concrete structure, which can be used by CM suites as input, is available.

Model transformation

Through the concept of model transformation, a model can be transformed to whatever is needed. By transforming a model to another model, it is possible to write one basic model for the development environment, which is then mapped to a specific model for change control and one for requirements management. Furthermore, the model-to-code transformation could also create Java classes and SQL create tables of a previous defined model for CM.

4.3 Software Configuration Management with Model Driven Configuration Management - the prototype

This section shows how to define a software configuration using models. An Eclipse installation will be established by first developing a metamodel as a foundation and afterwards writing a model representing a concrete configuration. This metamodel will not be able to support all possible software configurations, it's just an example for this case, in order to present this approach.

The first step is to write the metamodel using a domain model. Even the origin of the domain model is domain driven development, it fits well for this task. The different CIs, which are relevant for managing an Eclipse configuration have to be determined. There is one development platform entity and one entity for the programming language used by Eclipse. Since an Eclipse application might use Plug-Ins, these have to be included too. And after all, a programming language like Java will use additional packages and libraries, which are also important for describing the Eclipse configuration, since Java is an essential part of most Eclipse configurations. Therefore, it is obvious to model the Java configuration at an adequate level of detail. A domain model describing all this considerations might look like this.

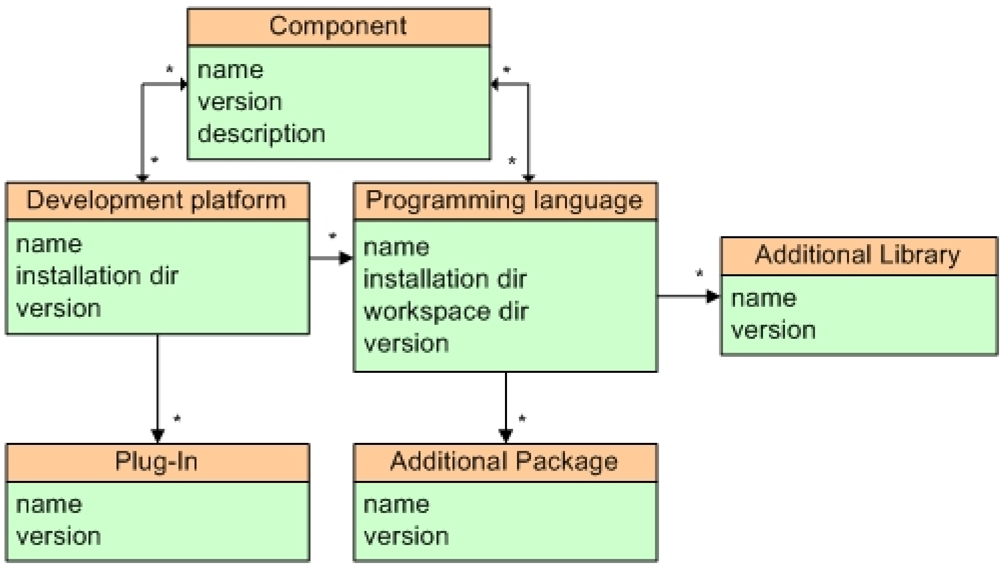


Figure 4.1: A domain model for development environments

Based on this metamodel, it is now possible to write concrete configurations. As told above, an Eclipse configuration will be modelled, but it is also thinkable to model a .Net development environment as well. This is one of the benefits using models to describe the foundation of a software configuration. Not only that different configurations can be created by using the model, it also provides a formalized standard which can be used by other applications. The concrete configuration for the Eclipse environment is shown by the next diagram.

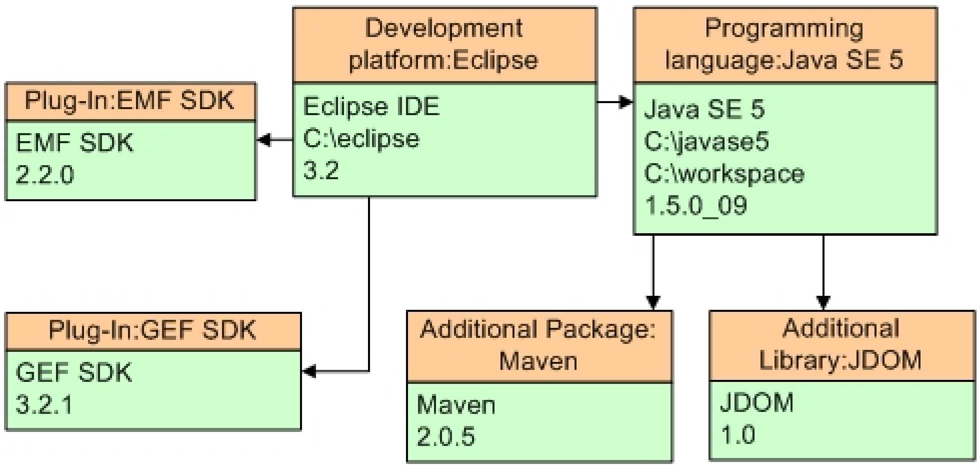


Figure 4.2: An Eclipse configuration model
(This model defines an Eclipse configuration using the domain model from figure 24)

Figure 4.2 illustrates a model configuration of an Eclipse environment. It includes every needed CI to manage an Eclipse installation and the model therefore provides all necessary information. In order to be able to use it as an input for CM tools, a transformation of the model to an XML file might be required. An XML representation of this Eclipse configuration model might look like the following listing.

```
<?xml version="1.0" encoding="UTF-8"?>
<softwareconfiguration>
  <Development_Platform
      name="Eclipse IDE" installation_dir="C:\eclipse"
      version="3.2" programming_language="Java SE 5">
  <Plug-In name="EMF SDK" version="2.2.0"/>
  <Plug-In name="GEF SDK" version="3.2.1"/>
  </Development_Platform>
  <Programming_Language
      name="Java SE 5" installation_dir="c:\javase5"
      workspace_dir="c:\workspace" version="1.5.0_09">
    <Additional_Package name="Maven" version="2.0.5"/>
    <Additional_Library name="JDOM" version="1.0"/>
  </Programming_Language>
</softwareconfiguration>
```

In the listing above, some relations of the model are mapped to elements and children of elements (development platform and Plug-In) and other relations are mapped as references (development platform and programming language). There are no rules defined how to map the model to an XML file, it depends on the needed input format of the XML file. Since the reason for representing the model as XML model is to have a possible input for a CM tool, the transformation of the model and the structure of the XML file are adapted to meet the requirements of the CM tool. So this listing is just an example to show, how such an XML configuration file might be structured. The next section deals with approaches to add MDCM to CM tooling applications.

4.4 A possible tool support

Complex but still efficient configuration management requires support of CM tools. In big projects with a CM process including change control, requirements management, versioning and release management, the use of a CM suite gets indispensable. That means, if tooling support is not available, MDCM has only theoretical relevance. In consequence of this fact, getting tool support and thinking about ways to include MDCM into common CM tools is a really important task.

However, this isn't the only issue to deal with. In order to get an advantage of using MDCM compared to common CM, MDA has to be included into our considerations. The models for CM have to be used for model driven development as well. This way the development and management approach will be the most efficient.

Suppose that some tools start supporting models for defining configurations and for providing a foundation for MDA and assume too, that each vendor of such a CM suite creates his own metamodel specification for those models. So another important step is to define an MDCM language, a set of metamodels and rules, which describe the structure and components of the models used by MDCM.

4.4.1 The perfect tool integration

Even there is no support for MDCM at the moment, to get an idea how useful MDCM can be, the perfect tool integration for MDCM is described in this section. Let's take a common CM tool, which supports all needed CM tasks. In order to support MDCM, this tool should include writing and editing models for MDCM. This includes writing domain models just like class models. Additionally, to offer variable models and mapping of models to different views, the CM tool should also provide a model-to-model transformer. All metamodels should be based on a standardized MDCM language, which provides definitions for modeling requirements, releases and software components as well as for application structures. All CIs and their relations are defined by models.

These models should also be used as an input for model driven development. One possibility is to include an MDA tool into the CM suite, another way to use the models is to provide the CM models as an input for the MDA framework. Rational ClearCase/ClearQuest already include the creation of models and the versioning of them. Also transformation and MDA are available, but the use of models for defining CIs and dependencies is not supported. Such a CM framework, which provides editing and writing of models to define CIs, their transformation and use for control, release and requirements management enables a whole set of new potentials for managing configurations as well as combining these configuration definitions with MDA.

5 Research Issues

This chapter deals with the research issues, which arose during the work on Model Driven Configuration Management (MDCM). To solve these topics was very important for the success of MDCM and therefore the crucial issues are explained next. The whole creation process of MDCM and the related research questions are divided into 4 phases, each one described in an own section. Section 5.1 deals with the fundamental concepts behind MDCM and the problems and challenges resolved in this context. Section 5.2 is all about the necessary research for the development of the procedure model. Section 5.3 describes the research topics of developing the supporting editors of MDCM and 5.4 deals with the creation of the case study. At the end of this chapter, the main efforts for establishing the MDCM are summarized in section 5.5.

Before the research issues are explained more precisely, the main motivation for the creation of MDCM, namely the following problems, are listed.

Summary of the motivation behind MDCM

MDCM has been developed to solve these problems:

- The redundancy throughout the whole development process
- The unused synergy effects between the development phases
- The inefficient collaboration of configuration management (CM) and software development (SD)
- The error-proneness of creating and editing XML configurations with usual editors (without graphical support)

5.1 Create a Concept

This section describes the details of the concept behind MDCM. In order to produce a target oriented realisation, the whole concept is focused on solving the major problems which had been the reason for starting to think about MDCM.

How to decrease redundancy throughout the whole development process?

It is well known, that the phases of a common development process have many similarities. Analysis, design and implementation all use the same informations and represent the same system. Only the intentions and the perspectives on the target system are different. For an example if the target system is a truckage company, then a truck will have to be represented in each phase. In analysis, the truck will be represented as a part of a use case diagram. In design he will be a class in an object model and in implementation he will be a Java class with attributes and methods. But the object, namely the truck, will stay the same. That's one fact, Model Driven Software Development (MDSD) is built upon. MDSD is already an approach decreasing redundancy by modeling a target system from analysis to implementation and by providing transformation processes between the different artefacts and phases. But MDSD has its focus only on the development and ignores the support processes and related management disciplines.

A usually very closely related management discipline is CM. It is in charge of managing the artefacts of SD by watching their versions and changes and much more. The definition of those so called configuration items (CI) is based upon the models of SD. MDCM has the main goal to use this relation to decrease redundancy and to improve the connection between CM and SD. In order to accomplish this, MDCM introduces a continuous tool support from the beginning of the analysis to the generation of the implementation. The idea is to create a foundation for an efficient and redundancy minimized SD process through flexible, end-to-end tooling. And therefore, the known MDSD approach is enhanced by including CM components. This also includes configurations of the application itself, which might be configuration files for the server of the application or configurations defining the building process and many more.

How to integrate CM to SD?

One very important question for MDCM was how to improve the collaboration between CM and SD. MDCM adds CM to the SD procedure model so that it becomes a part of it. There are many ways to do this, from modifying existing procedure models to creating a completely new approach. But since there are already established and approved approaches for CM as well as for SD, the MDCM procedure has been created on those existing methods.

It was also a big goal to benefit from the close relation of some CM and SD artefacts. In order to do this, general models have been developed, which are able to provide a foundation for both, CM and SD models. The whole approach is based on MDSD. Figure 2.3 in the chapter 2 From traditional Software Development to Model Driven Configuration Management illustrates the different artefacts and their relations. As can be seen, the MDSD process has been enhanced by CM artefacts and therefore MDCM can benefit even more from the features of MDSD.

By using the MDCM approach, the efficiency and quality of the collaboration between CM and SD will rise. Because in the MDCM process, the configuration manager and the software architect will cooperate closer and so the coordination will be better than in traditional SD.

The level of automation?

Although the basics of the approach behind MDCM had been defined, there was still an important open issue: the level of automation throughout the process. There are already some established, well working realisations of MDSD which provide the foundation for MDCM. MDCM has its main focus on generating the implementation of the application and the input for the CM tooling.

Another possible automation are the transformations between the different models of MDSD. But unfortunately it is very complicated or even impossible to define the necessary logic behind these transformations. Therefore and because of the relatively small benefit, MDCM doesn't work with those model-to-model transformations.

5.2 Develop the procedure model

It has already been decided, that CM should become a part of the software development process and that it has to be included to the procedure model of SD. The concept also dictates that the MDCM procedure model has to be based on existing and established approaches. The major decisions of this phase are now which SD model will be selected, how to include CM to the process and how to integrate the MDSD approach.

The choice of the SD model

There are many procedure models, which have been considered. The following requirements have been set up to ease the decision:

- not too complex or detailed, because that would complicate enhancements of it
- flexible enough to integrate parts of configuration management
- approved and state of art

Additionally to those points it would be helpful if the procedure model already integrates MDSD.

There are many different models, which had been considered during decision making. Since the selction of the procedure model is closely related to the specific model and because of the fact, that there isn't one model which does everything better than the others, MDCM tries to build upon the concepts all procedure models have in common and enhance them. This approach is not focused on a specific model. Instead basic concepts and processes are enhanced. So MDCM has to be seen on the same level as MDSD.

Integrate CM

Since it had been decided to use MDCM as an approach like MDSD, the last important question is how to integrate CM to MDCM. The first consideration was to include it to the procedure model. But because CM is a support management process, it has many more iterations and a different build-up than an SD procedure model and CM therefore doesn't really fit into it. Additionally it had already been decided, that MDSD should work with different procedure models. An own procedure model for CM which might support the selected procedure model is also not an ideal approach, because CM has no complex workflows and doesn't necessarily need a procedure model to work fine. Therefore it has been decided to apply CM after established best practice methods including our MDSD enhancements. Section 3.1.4 Parts of Configuration Management describes such an approach.

5.3 Build the editor

A big, important part of MDCM is the editor, which should provide optimal tool support throughout the MDCM process. Significant research issues have been how to create an editor offering the demanded features (flexible, continuous, extendable, efficient) and the selection of the technologies providing the foundation for the Model Driven Configuration Editor (MDCE).

Technologies of the MDCE

One basic requirement for all used technologies was that they have to be open source technologies, because the whole project had been designed as an open source project. The editor also has to support the model driven approach, since it is one fundamental part of MDCM. So at the start the first challenge was to select a proper open source MDSD framework. There was a relatively small offer and so the decision felt on oAW (see section 3.4.1), which provides enough tutorials for easy access and enough functionality for our purpose. The use of oAW for MDSD is also the subject of the master thesis of Benedikt Weismann [Weismann 2006]. The next challenges were to select a framework to create and write models and one which provides the graphical representation of them. A very new and promising technology was the GMF, which offers comfortable access to EMF models. After a short review of GMF, it has been decided to build the editor with the GMF and the EMF (see section 3.3.2 and 3.3.3), because both provide the needed functionality in an efficient and easy to learn way. All documents and files of the MDCE and also the connection from the MDCE to SD and CM tools are based upon XML. Therefore and as a support for oAW, JDOM (see section 3.4.2) for XML manipulations has been included to MDCE. JDOM can handle some XML transformations better than oAW which needs to look at them as model transformation. Now that the technologies had been decided, the next step was to plan the development of the editor.

Developing MDCE

There are very high requirements for MDCE. As already mentioned, it should be flexible but also efficient and continuous. The creation of an editor fulfilling all these requirements is extremely complex and time-consuming. Additionally it is not always possible to match the requirements of a project, because individual project requirements need individual solutions and also individual tooling. As the development of an editor based on GMF and EMF doesn't take long and also the creation of the oAW and JDOM implementation is not that complex, it has been decided that the tool support for MDCM shouldn't be realised as an editor but as an approach how to create an editor for the described purposes.

So MDCM offers a description how to generate an MDCE editor. It's a kind of best practise manual for GMF and EMF combined with oAW and JDOM. Additionally, there are already developed editors which provide basic functionality and which can be enhanced. This approach makes it possible to create an editor matching individual requirements for every need.

5.4 Perform a case study

The last important research issue has been how to build an efficient case study to review the solutions of MDCM. It has to be shown, if MDCM really is able to deliver a benefit in direct comparison with traditional SD. Additionally the case study should also provide information on the usability and features of the MDCE.

The whole case study should be based upon a small real world project, which has to be partly redeveloped by using the MDCM approach and its editor. Fortunately, a small project called flightplan which is a part of the CEP has been found. It has the right size and complexity to provide an excellent foundation for the case study.

As a result of the case study, the work and time effort will be measured and compared with the estimated effort of the traditional SD approach. On the foundation of these results it is possible to decide if MDCM is a good working solution or not. The case study can be found in chapter 7.2 The flightplan example.

5.5 Main efforts to establish Model Driven Configuration Management

This work already described the concept of MDCM and the benefit it could deliver. But of course, there is also a lot of work to do to establish MDCM, especially since there aren't already tools which offer all functionality for MDCM. Next, this work shows a list of tasks which have to be done to introduce MDCM to a project.

- Create MDCM Metamodels: Due to the fact that metamodels for configurations aren't already available, they have to be defined for all configurations managed by MDCM. This is only necessary as long as no metamodels are at hand.
- Write models: Based on the metamodels, models of the configurations can be created. This is not an additional effort, since the definition of configuration items and their relations have to be done whenever CM is applied.
- Add models to the CM suite as definition of the CIs: This step is the main problem. Since no CM tool offers an import of models describing configurations, a public licence tool might be modified to provide such functionality. Another way is to develop a CM tool which offers the needed functionality. Both approaches are complex and very time-consuming
- Modify/Transform models for MDA: Since some CM tools (ClearCase for example) already provide MDA support, it's not much additional work to transform the configuration models for MDA use.

6 Implementing the Model Driven Configuration Management Components

Based on the research issues and the resulting decisions, this chapter describes the work with Model Driven Configuration Management (MDCM) and its concrete application area. This chapter realises the ideas and approaches researched in the previous chapters. MDCM can be seen as two mostly independent components interacting with each other. When looking at the research issues and goals of MDCM, they can be classified in two categories.

The first category is concerned with the process of software development (SD) using the model driven configuration approach. It describes how configuration management (CM) and SD are integrated in one process and explains the procedure model and most of the theoretical aspects. Section 6.1 Software development using Model Driven Configuration Management describes, how to include the different approaches of MDCM to a software development project. Each important stage, its artefacts and enhancements through MDCM are explained.

The second category is all about applying the MDCM to a project. It defines how to create editors supporting the MDCM process, how to transfer models efficiently to configuration files and addresses the subject of tool support. Section 6.2 The Model Driven Configuration Editor in action is all about the Model Driven Configuration Editor. It describes how to create an MDCE for individual usage in detail.

6.1 Software development using Model Driven Configuration Management

This chapter describes the enhancements of software engineering through MDCM. The main intention behind this chapter is to show how to integrate MDCM into the software development process. MDCM doesn't refer to a specific procedure model in order to keep it as general as possible. Instead the different stages of software development, which can be found in every procedure model, provide the foundation for the integration of MDCM. The stages essential for MDCM and a short description can be found in section 2 From traditional Software Development to Model Driven Configuration Management.

For all these phases, a common proceeding and a list of artefacts exist. The idea of describing the different artefacts for each phase is taken from [ZUSER et al. 2001]. For a software development process using MDSD, the proceeding and the generated artefacts differ from the ones of a common software development process. In MDCM, configuration management has been added additionally to the software development process. Thereby the proceeding and also the artefacts changed. The next section describes the differences between a traditional software development (SD) process and one using the MDCM approach and shows how to integrate configuration management better into the SD process.

6.1.1 The merge of Model Driven Software Development and Configuration Management

A very fundamental goal of MDCM is to enhance the connection between MDSD and configuration management (CM). The important features of MDSD are the representation of information as models and the transformation of these models from model to model or to code. Section 3.2 Model Driven Software Development describes MDSD in greater detail. Based on this idea, MDCM tries to describe configurations also as models and hence uses models to represent CM informations. Configuration items and their relations, which are explained in section 3.1 Configuration Management, are designed as models like the ones of MDSD. Section 3.3 Unified Modeling Language and Eclipse frameworks provides the necessary informations on technologies and methods used for modeling in general. Chapter 4 describes a prototype which presents the basic features of MDCM using a small example.

MDCM uses the methods of MDSD to define general models and enhance them with additional information through transformations. In order to decrease the redundancy of a software process, MDCM extends the approach of MDSD and includes configuration management. In MDCM, those parts of CM and software development which are related, use the same basic models as a foundation for further transformation and processing. Chapter 2 From traditional Software Development to Model Driven Configuration Management provides an overview on this approach through illustrations and short explanations.

Unfortunately this is not possible for all issues covered by CM. So an important aspect is to define basic models which provide the foundation for CM and SD. The configuration management modeling language is a set of metamodels, which provide the definition of the models of MDCM.

6.1.2 Configuration Management Modeling Language

The Configuration Management Modeling Language (CMML) is a set of UML metamodels describing the structure and rules for the models used by MDCM. These metamodels describe the basic models which can be used as a foundation for CM and SD. In chapter 4 4a development environment metamodel is introduced. This is a metamodel of CMML. The different metamodels are classified according to the stages of a software development process (described in section 2.1 The traditional Software Development process with Configuration Management). So there is continuous support by CMML through the whole SD process. The different metamodels of CMML can be found in the following sections 6.1.4 to 6.1.8. By using metamodels as a foundation for MDCM, all models used according to the CMML are standardized. Because of that it is possible to write a tooling which allows to easily write such models and to provide transformations. Before CMML, a general tool support for MDCM was not possible. As can be seen later, the MDCE can be used to provide a tooling for metamodels for MDCM.

Additionally the metamodels of CMML ease the entry to MDCM. They are an important part of MDCM and help to define its procedure. Each section related to a stage of the SD process will include metamodels of the CMML. The next section describes the procedure model of MDCM and the important roles.

6.1.3 Procedure model and roles

This chapter introduces a procedure model for MDCM based development processes. The Section MDCM procedure model describes an approach to combine MDA with CM to enhance the development process and the section MDCM roles deals with the different roles and their tasks concerning the MDCM process.

MDCM procedure model

Procedure models have already been explained for configuration management and software development. In this section, both models are merged into a new procedure model, which provides a foundation for applying MDCM. Figure 6.1 shows the result of merging the models and also illustrates the movement of the MDCM resources.

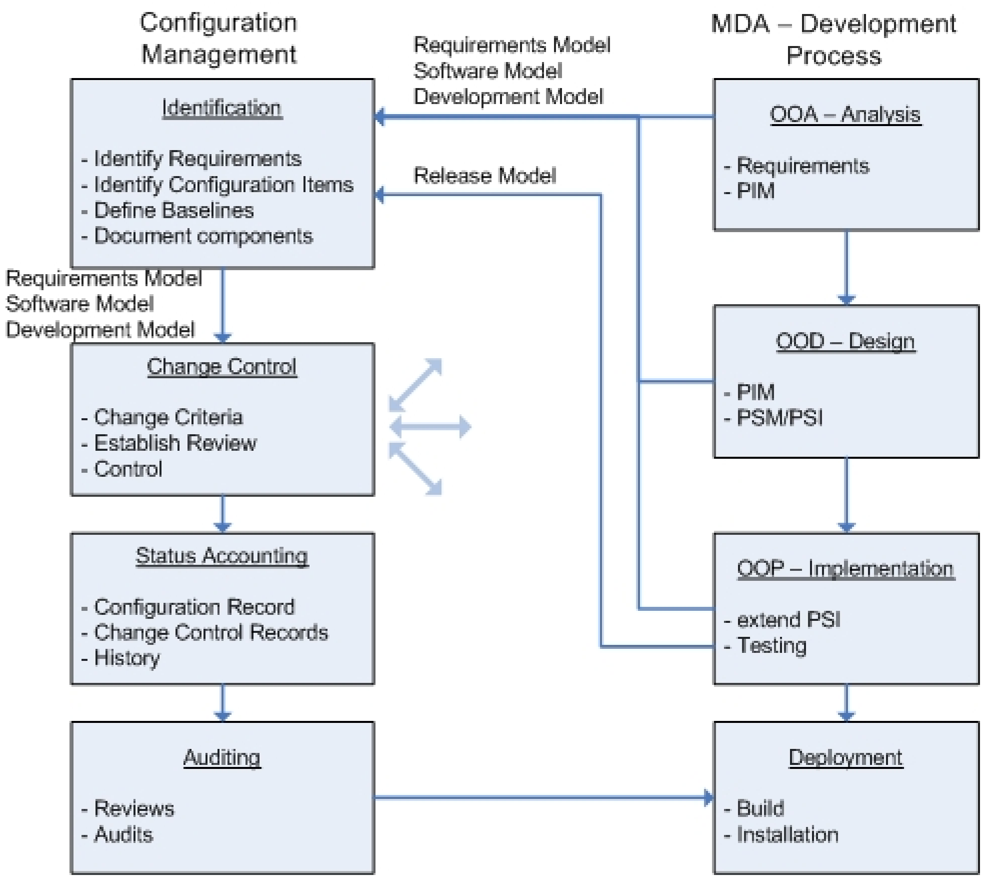


Figure 6.1: Model Driven Configuration Management procedure model

As can be seen, on the right side is the MDA process and on the left side the CM process. The arrows indicate the data and information flow between the different process items. At first, OOA has to be done in order to identify the requirements, the development environment (software model) and the different development artefacts (packages, classes, etc.). The models resulting of OOA are sent to the CM Identification phase. Based upon those models, the CM system is initialized and the CIs are defined. Change management is applied to each CI and the development continues with OOD and OOP. Both MDA phases are supported by change management and status accounting is done. Based on status accounting, audits and reviews can be provided to the MDA process. The MDA process ends with the deployment of the developed application. The CM process closes after status accounting has been done and all important audits and reviews have been generated and saved.

MDCM roles

Project manager

The project manager is a very traditional role in a software process. He is responsible for the success of a project. He has to coordinate the project team, the customer and manage the whole process software development. In MDCM, he is in charge of introducing MDCM to the project team and integrating it into the software development process.

Software designer/analyst

The software designer has to define the software and plan a problem-solving software solution. The software analyst has to prepare the software requirement documents. Both roles have to deal with the MDCM-specific requirements and software model. They also have to create the design of the PIM. The following artefacts have to be created or modified by the role software designer/analyst:

- Requirements model
- Design model
- PIM design
- PSM design

Software engineer

The software engineer is the one to develop the application. In this case it means to help designing the PSM/PSI mapping and to extend and enhance the generated PSI. The software engineer deals with the following artefacts:

- Development model
- PIM/PSM/PSI mapping
- PSI manual development
- Deployment model

Configuration manager

Additionally to his usual work, the configuration manager has to provide the import of the different models of MDCM to the CM system. This is a very important task since it is crucial for the success of MDCM to import the models to the CM system.

The tasks of the configuration manager include:

- CM control
- CM tool integration
- Change management

Figure 6.2 illustrates the MDCM process including the described roles.

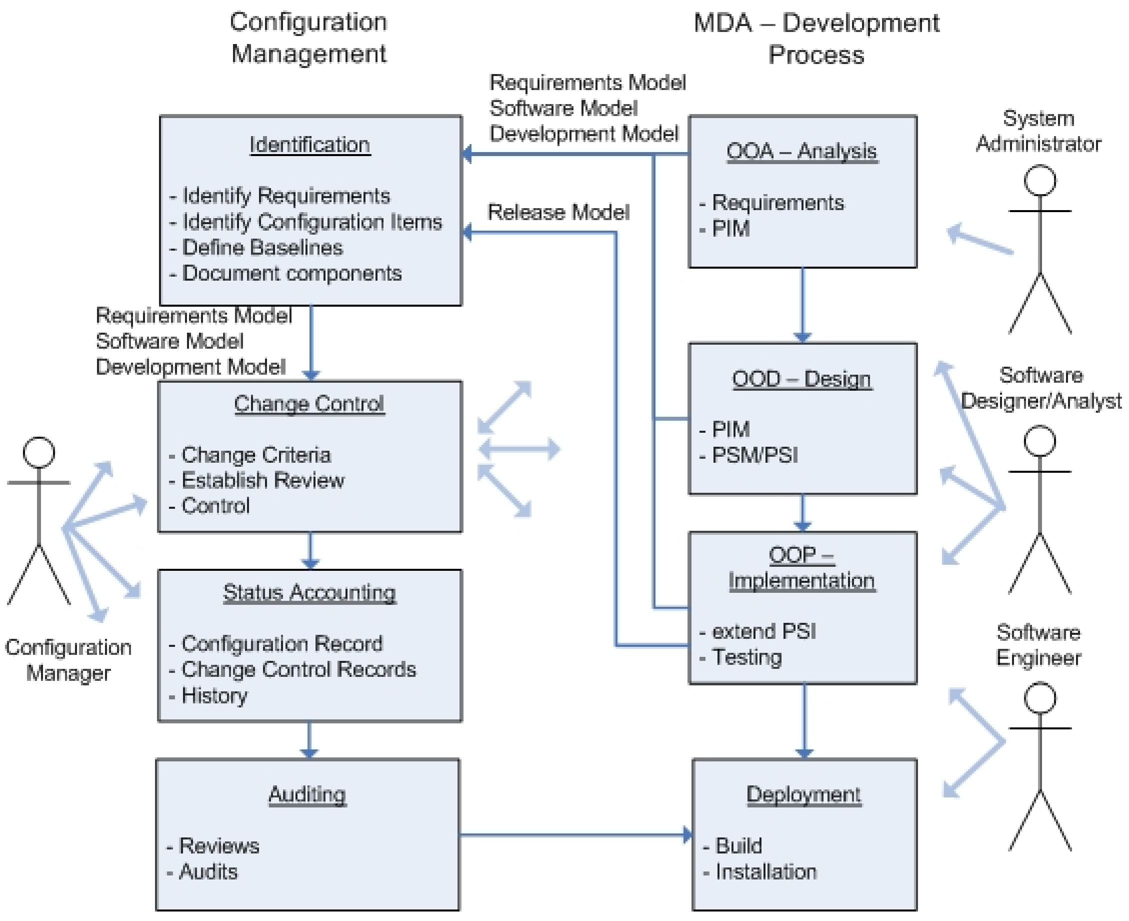


Figure 6.2: Model Driven Configuration Management process including roles

6.1.4 Analysis

The analysis stage provides the foundation for the whole software development process. The target of this stage is to define the requirements of a project and to document them as good as possible in order to enable a quick and effective design stage. Some crucial decisions regarding the structure and organisation of the documentation and of the project have to be made. Configuration management is initialized and the project directories are created. It is recommended to already establish a document management system in this stage and to include all created documentation into it.

Beneath the organisational challenges the actual creation and writing of the requirements is most important for the further software development process. Each procedure model and software development approach specifies different artefacts for each stage. In MDCM, the most useful artefacts are described and defined by metamodels. The next section shows a list of the important artefacts of the analysis stage, which altogether can be referred to as requirement documents.

List of artefacts

- project description

  The project description defines the aspects of the project itself. It defines the scope of the project, the procedure model, the iterations and much more.
- system description

  The system description defines the most important aspects of the target system. It specifies the initial situation, the goals and boundaries of the target system and of already existing systems.
- use case diagram/description

  The use case diagram shows the different roles interacting with the target system and their activities. Use cases provide the basic informations for most development processes. The Use Case Diagram is a part of OMGs UML [1]. A good introduction can be found at www.parlezuml.com. These use cases can be further specified by a use case description, which defines the details of the use case.
- conceptual class diagram/domain diagram

  The domain diagram describes the static components and its relations of the target system. Depending on the stage of the software development process, the class diagram describes different aspects of the system. In the analysis stage, the conceptual class diagram describes the main components and structure of the target system. In contrast to the class diagram of the design stage, this diagram doesn't apply object-oriented aspects and is not concerned with the actual implementation [2].
- business process model

  The business process model describes the processes important for the target systems. In this case, a business process further describes the activities of a use case. If this business process is supported by software, it can also be called workflow. There is a business process modeling notation BPMN [3] which defines how to model a business process. Often business processes are also described using the UML activity diagram.

Most of these documents are connected to each other. In order to keep the documentation consistent, these relations have to be defined and the redundant informations have to be maintained. The Configuration Management Modeling Language defines a metamodel for this purpose.

[1] Further Information can be found at http://www.uml.org/

[2] A good tutorial for several UML diagrams can be found at http://www-ivs.cs.uni-magdeburg.de/~dumke/UML

[3] http://www.bpmn.org/

CMML for the analysis stage

There are different UML and textual documents which describe the requirements of a project. The different documents are defined as configuration items and their dependencies as connections between the CIs. The requirements model provides an overview of all requirement documents and specifies the relations between them. An MDCE editor based on this metamodel can be created in order to ease the interaction with the requirement model. More informations on using the MDSD approaches to transform the CMML models can be found in chapter 7.2 The flightplan example.

The metamodel of the CMML for defining requirements as CIs is illustrated below.

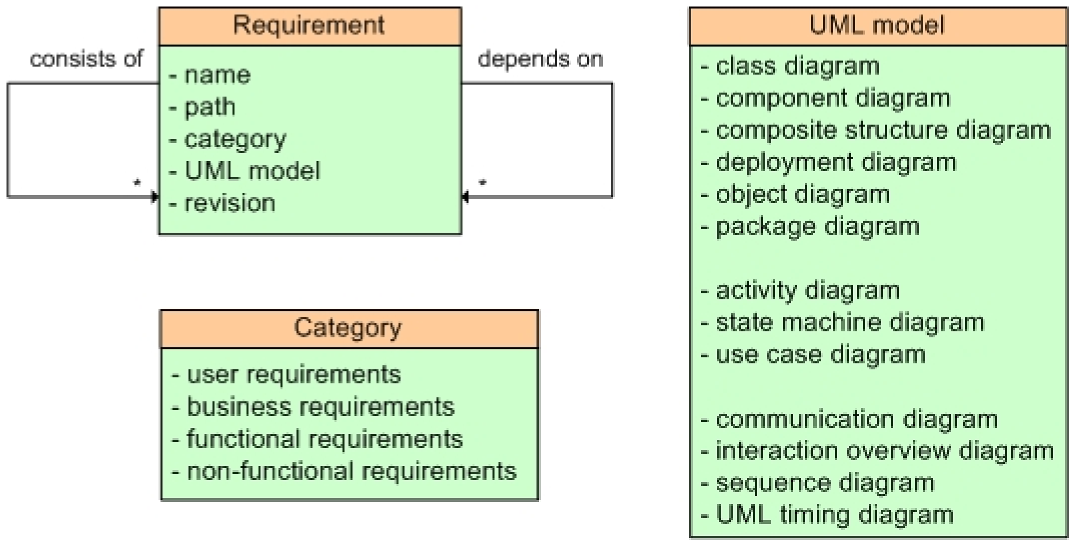


Figure 6.3: Metamodel for Model Driven Configuration Management requirements

Each configuration item of this stage is modeled as a Requirement element. This element can consist of and also depend on other Requirement elements. Each CI has the attributes name, path, category, revision and UML model. The attribute path defines the local path in the CM repository (it is assumed that the CM application uses repositories) and the revision attribute its current revision number. These two attributes should be maintained by a CM tool, so that they always have the right values. Additionally, the change management tool should work with the dependencies defined between the requirement CIs.

Figure 6.4 shows a concrete appliance of the requirements model. When using the MDCE, the model can be transformed into an XML document which could then be used as an input for various further processings.

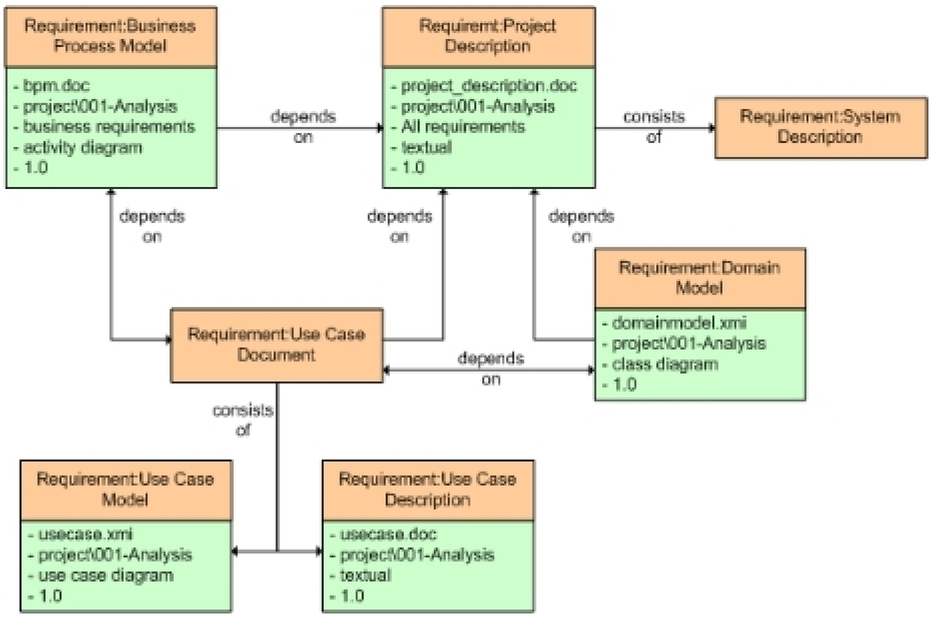


Figure 6.4: Requirements model based on the Configuration Management Modeling Language
)

Of course, MDSD can be applied in many ways. But there are best practice approaches and processes which have been established as state of the art. Each stage has short section showing the relations of the artefacts and of MDCM to the MDSD.

Analysis in MDSD

During the analysis stage, the computation independent and the platform independent model are created. The first step is to create the CIM. The domain model described above provides the basic informations. On the foundation of the domain model, use cases are created and the business processes are described. Activity diagrams are used to describe the data flow and operations. These diagrams have to be designed using MDSD tooling in order to be able to transform them into a platform specific model later. The requirements model of MDCM provides an overview model of the different MDSD items.

6.1.5 Design

The design stage deals with the realisation of the analysed requirements. Based on the knowledge gained during the analysis stage, the target system is specified. The structure as well as the behaviour are defined. Class diagrams and database models will be created as well as activity and sequence diagrams. Also detailed workflow descriptions can be modelled in the design stage. The level of detail of the models and documents depends on how complex the target system is. Complex systems demand a detailed design.

In addition to the description of the target system, organisational challenges have to be met. Database- as well as application- and other necessary servers have to be available and configured. Also the development environment should be configured. It is one goal of the design stage to do all required activities to enable an immediate implementation start after it.

List of artefacts

- architecture description

  The architecture description defines the structural and technical aspects of the architecture of the target system. To choose the right architecture is most crucial for the success of a software development project. There are many predefined and established architecture pattern, so that most times it is not necessary to invent a new architecture. Architecture descriptions are often complemented by models based on the UML deployment diagram [4] or the various architecture description languages like ACME [5].
- class diagram

  The class diagram in the design stage is a detailed implementation focused enhancement of the domain model of the analysis stage. Unlike the domain model, it really describes classes in terms of object oriented classes and defines the structure of the implementation.
- database description

  A detailed database description including a database model and a database definition have to be created in the design stage. Usually an ERD (Entity-Relationship-Model) or EER (Extended-Entity-Relationship-Model) are used to model the structure of a relational database [6]
- behaviour model/workflow description

  Complex workflows in the target system have to be further described and defined then the business process model does. The behaviour model is a collection of different models/diagrams like the activity or sequence diagram and textual descriptions. The goal of the behaviour model and of the workflow description is to describe the data and information flow of the crucial processes as far as necessary to allow a clean implementation.
- prototype implementation

  A prototype implementation demonstrates the system architecture, user interface and basic functions of the target application. The prototype will be presented to the customer in order to show the status of the application. A prototype makes sure that the planned application will meet the customers requirements.

[4]More Information at `http://www.uml.org/`

[5]For further Information visit the ACME Project homepage, `http://www.cs.cmu.edu/~acme/`. Many interesting papers about the software architecture can be found at `http://www.bredemeyer.com/papers.htm`

[6]A good overview of the EER is provided at `http://www.cs.man.ac.uk/~horrocks/Teaching/cs2312/Lectures/Handouts/Eermodelling.pdf`.

CMML for the design stage

The artefacts of the design stage are closely related to the artefacts of the analysis stage. Due to this fact, the metamodel of CMML is also similar. The design meta model not only shows the relation between the design artefacts, it also show the relation to the artefacts of the analysis stage. Hence it is possible to track dependencies over the boundaries of the development stages.

The metamodel of the CMML for the design stage is shown below.

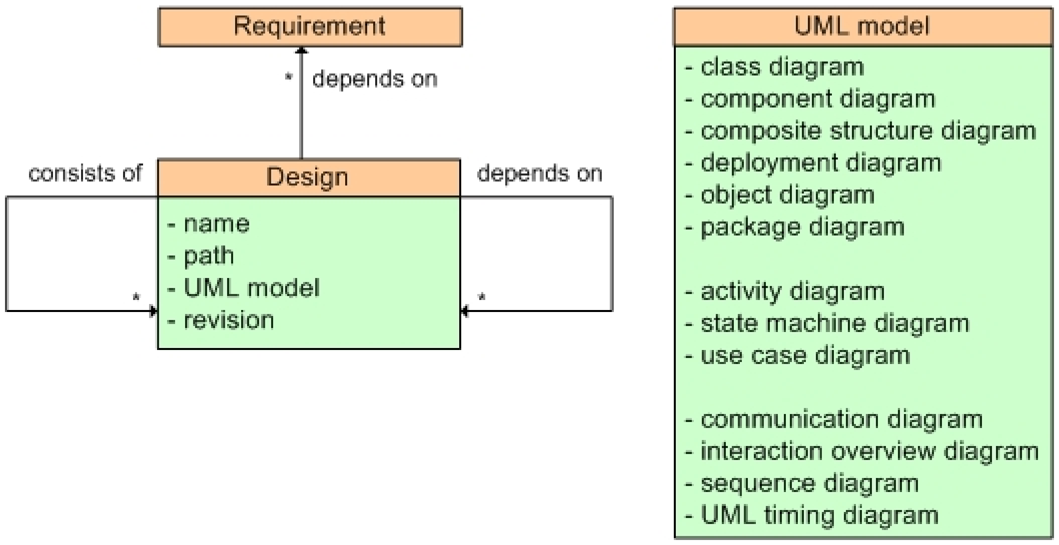


Figure 6.5: Metamodel for Model Driven Configuration Management design

Each design artefact can depend on or consist of other design artefacts. Additionally, the design artefacts can also depend on requirement artefacts. A model based on this metamodel is able show all connections between the different artefacts and thus provides enough information to prevent inconsistency between the different documents and models.

A possible design model is illustrated by Figure 6.6.

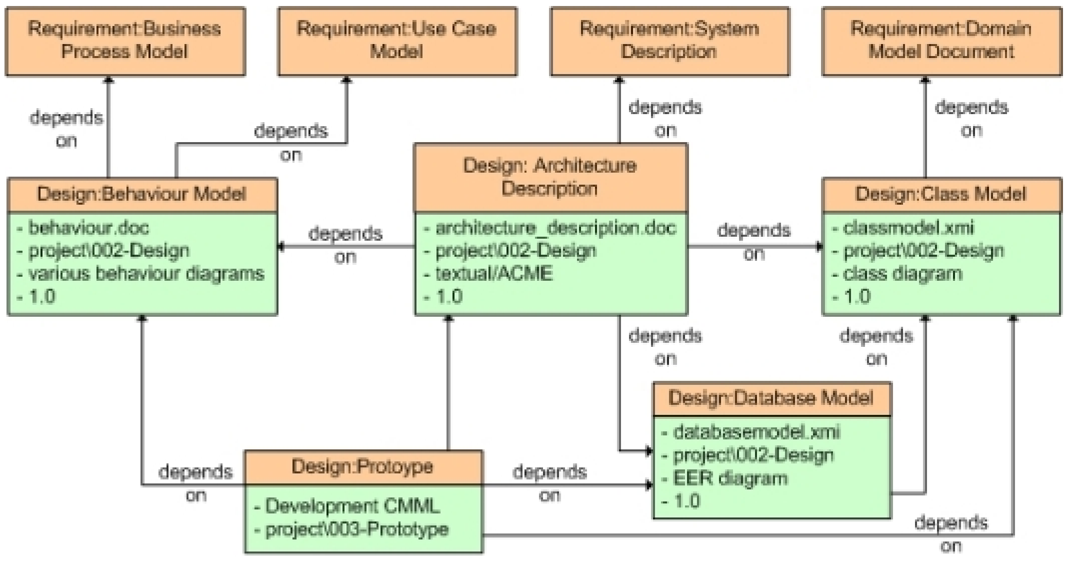


Figure 6.6: Design model based on Configuration Management Modeling Language Design

The prototype artefact in figure 6.6 is pointing on a development model. The prototype is small application and can be represented by a CMML implementation model which is explained in the next section.

Design in MDSD

At the beginning of the design phase, the platform has to be modeled. This means, platform specific transformers have to be developed, which generate the platform specific model. A possible and common transformation is from the domain diagram to the class model. The transformer will add the platform specific and object oriented informations to the diagram model and generate the class diagram. Most of the behaviour models have to be manually transformed from the analysis stage to the design stage. But it is possible to use transformers to generate code from the behaviour models in the implementation stage.

6.1.6 Implementation

After the design documents have been written and the development environment including the needed resources had been set up, the implementation stage can start. At the beginning the development project has to be created and included to the document management system. If not available, coding conventions and important programming patterns have to be defined.

In an MDSD process, the actual development starts with generating the implementation skeletal structure and the database tables and definitions of the implementation. If modelled proper, even parts of the flow and the operations can be generated out of the behaviour models like the activity diagram or sequence diagram. After that, still a large part of the application has to be developed manually. Additionally it is important not to forget, that also code has to be documented.

List of artefacts

- coding conventions

  Coding conventions define the style of the programming and make sure that the implementation is structured and consistent, even when many developers are working on the same application. Furthermore coding conventions may also define pattern, which guide the developers through different well known programming challenges.
- database definition

  The database definition is a collection of statements to create the database and the tables including their relations and columns. Those statements are written in the database definition language DDL. Most times this artefact is generated out of the database model.
- technical specification

  The technical specification is the central documentation of the application code. Specific features and complex code snippets are described in detail. Central important database access and service calls are documented. The main goal of the technical specification is to enable programmers to understand the code of an application without the need of further informations.
- operating manual

  The operating manual specifies the different system environments the application is developed, tested and deployed on. The operating manual describes the different servers and the distribution of the application on these systems.
- application

  The application itself is more a collection of artefacts than a single artefact. Because of this fact and because of its importance, the application is described in a separate metamodel.

CMML for the Implementation Stage

Because of the importance and complexity of the implementation stage, there are three hierarchic levels of the CMML implementation model. The first and general view on the implementation artefacts called artefact view is the same as used for the design and analysis stage. The application itself is defined by two detailed views, the project and class view, which describe every important structural part of it. The most detailed class view enables the generation of code an CM item definitions and is thus the major model of the whole MDCM process.

Artefact view

The view on the basic artefacts of the implementation stage is illustrated by figure 6.7.

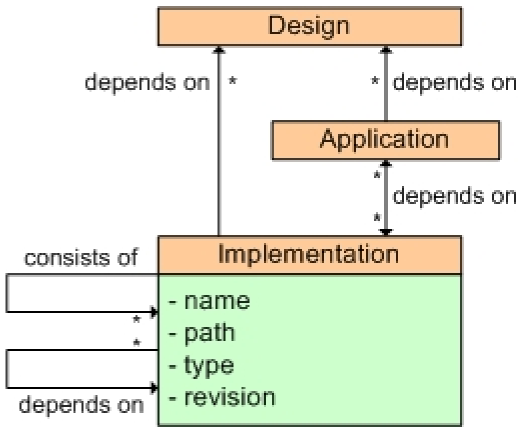


Figure 6.7: Metamodel for the Model Driven Configuration Management implementation artefact view

The artefacts of the implementation stage are separated into application and implementation, where implementation refers to the documentation artefacts and application means the actual program. The application is separated from the rest because it is described in detail by the two upcoming detail views. An actual implementation model in the artefact view is shown on the next page.

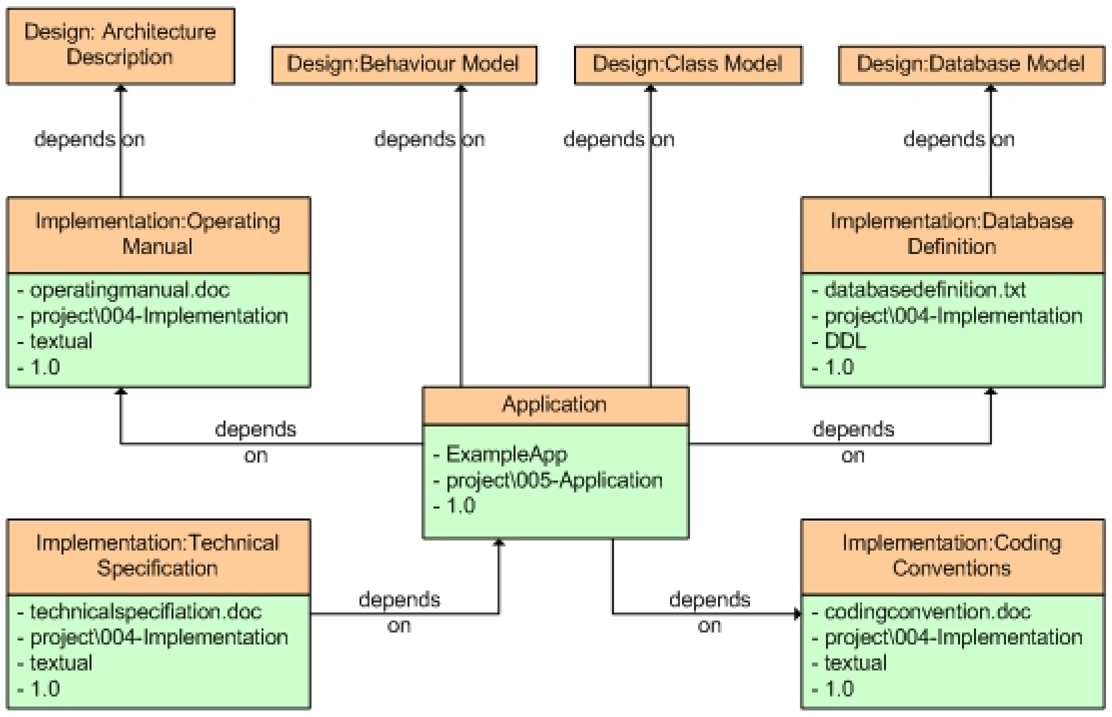

Design: Architecture Description
Design:Behaviour Model
Design:Class Model
Design:Database Model
depends on
depends on
depends on
depends on
Implementation:Operating Manual
- operatingmanual.doc
- project\004-Implementation
- textual
- 1.0
Implementation:Database Definition
- databasedefinition.txt
- project\004-Implementation
- DDL
- 1.0
Application
- ExampleApp
- project\005-Application
- 1.0
depends on
depends on
Implementation:Technical Specification
- technicalspecifiation.doc
- project\004-Implementation
- textual
- 1.0
depends on
depends on
Implementation:Coding Conventions
- codingconvention.doc
- project\004-Implementation
- textual
- 1.0

Figure 6.8: Artefact implementation model based on Configuration Management Modeling Language

Project view/Class view

Figure 6.9 illustrates the metamodel for describing the components and resources of a development project which is used for the project and class view. This metamodel is focused on describing Java development projects. It will also work as a foundation for projects using other development languages, but some modifications might be necessary. When using this metamodel for defining a model for a large application with many projects and a lot of resources and classes, the model will soon get very large and confusing. Therefore the model is separated into two different views.

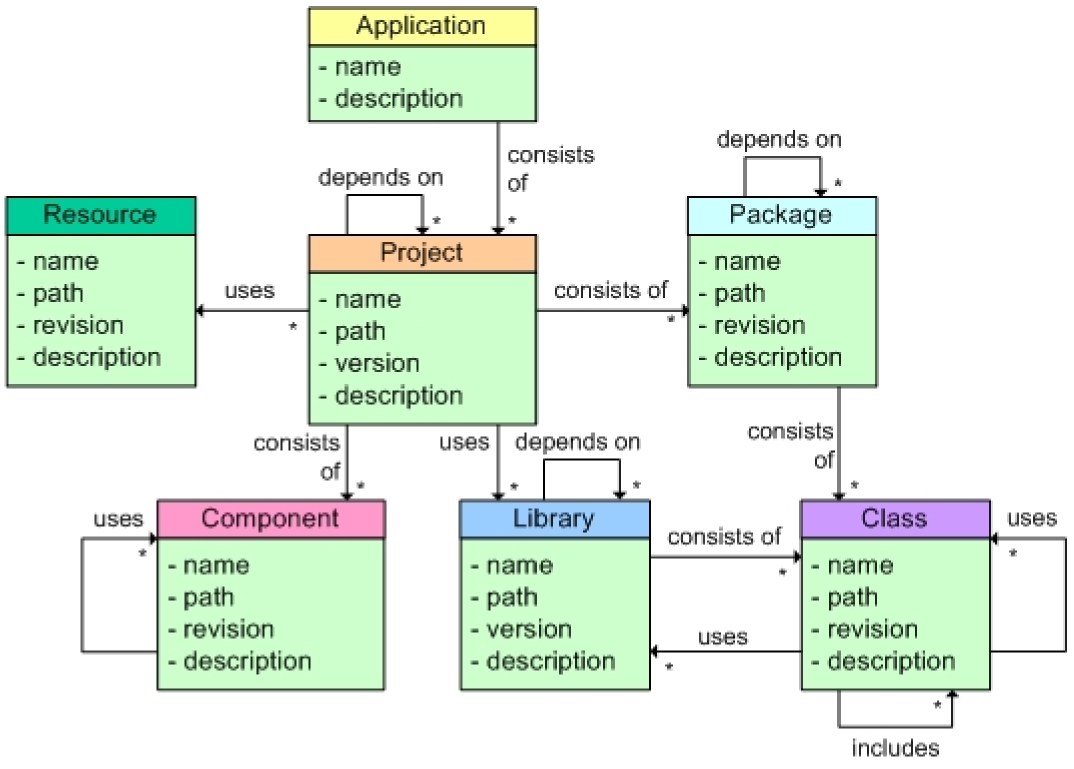


Figure 6.9: Metamodel for Model Driven Configuration Management development

The model in Figure 6.10 shows a project view of the Naiad example project. Each project and its relations to other projects are defined in this model. It already provides possible input data for change control. The next step is to write detailed models for each project. Figure 6.11 shows the model describing the JDOM project of the Naiad event server example. This model provides all necessary information for managing the project. It's very useful for everyone working with this project. It offers a clear view of the project and describes all CIs and relations. Thus it is a suitable input for change control, versioning and release management.

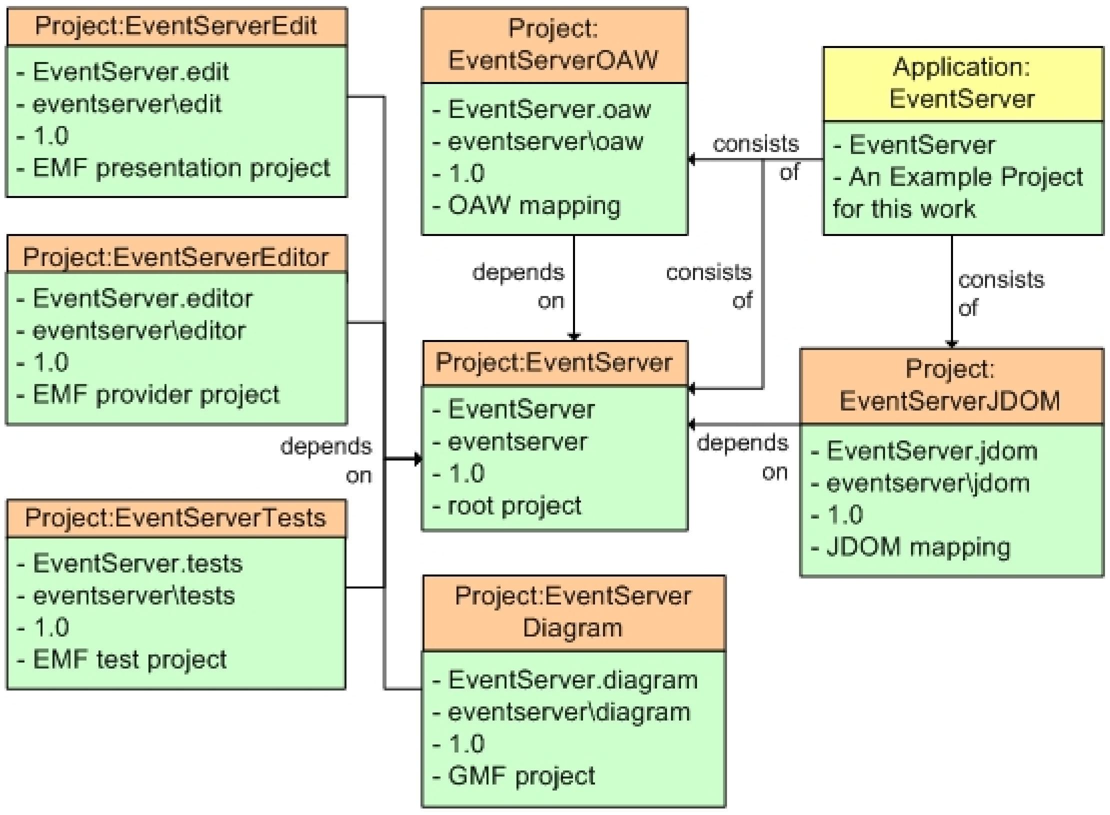


Figure 6.10: Model Driven Configuration Management project view of Naiad

It is very time-consuming to model each single component of a huge software system. Fortunately it is not necessary to add every small file or class to the model. It is absolutely sufficient to model only important CIs and group all other components.

The challenge is to add as few components as possible and still providing all needed information for configuration management. A concrete appliance of these models is shown in chapter 7.2 The flightplan example. This example also explains the possibilities to generate code and configuration definitions out of the implementation models.

Implementation in MDSD

At the beginning of the implementation stage, the platform specific implementation is generated. For this purpose, transformers have to be written again. Since the generation of code on the foundation of UML diagrams is widely spread in software development, there are many predefined generators or templates, which ease the code generation.

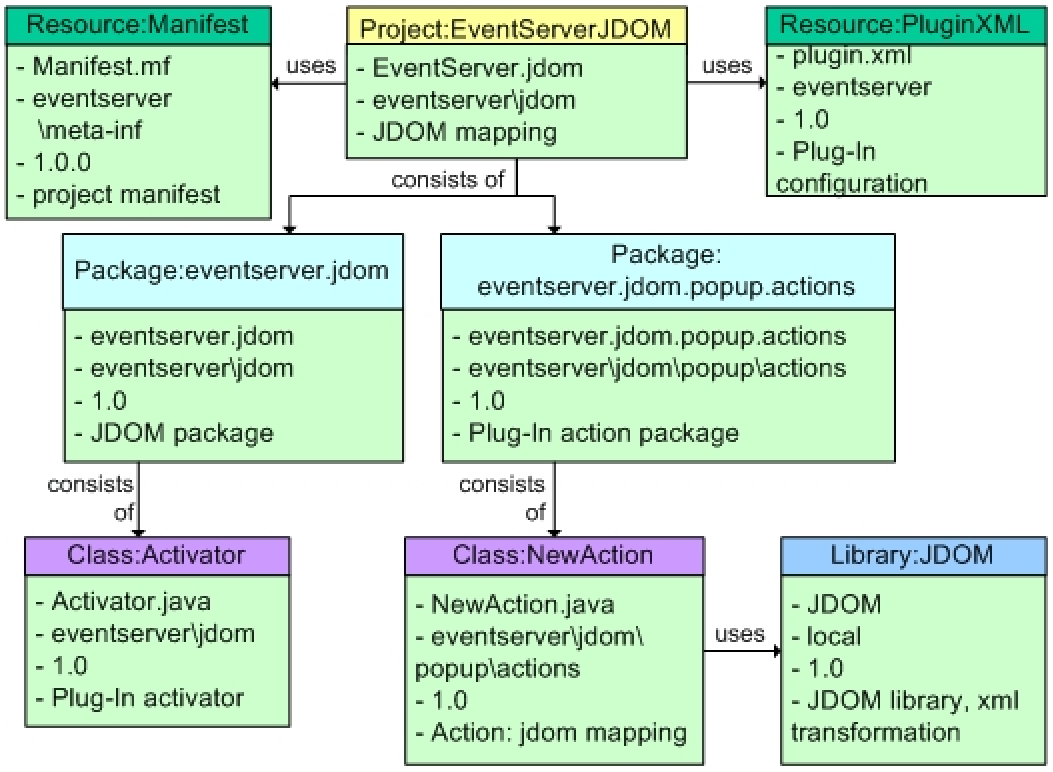


Figure 6.11: Model Driven Configuration Management JDOM project of Naiad

The class structure of the application including packages, constructors, getter and setter methods and attributes are generated out of the class diagram. Also the database tables and their connections are created on the foundation of the database statements generated on the foundation of the EER model. This is straightforward and not necessary only a part of MDSD because many EER tools support the generation of DDL statements. Additionally there are many approaches to also generate the DAO classes for the database access. That is in most cases very useful, since the standard CRUD (create, read, update, delete) operation are nearly always the same.

Not only the structure, but also the design logic can be transformed into code. However, the generation on code for business logic was and is the major weakness of the MDSD. In order to be able to generate logic workflows, they have to modelled in a great detail so that the implementation itself won't take longer.

Still modeling business logic is suggestive for complex workflows in order to minimize error-proneness and wrong logic adaptation.

Even code generation is able to create some parts of the application, the major part of the implementation stage is still manual coding.

6.1.7 Test

In the test stage, the implementation is tested against its specification. Reviewing a system is not only done by running some uncoordinated tests. Instead a test plan has to be created, where all important aspect are listed. Most times the test cases are created using the use cases as a foundation. In each test case, the application is reviewed on the fulfillment of the functional and non-functional requirements. The more errors that could be found, the more successful the test stage was.

The found errors are collected and documented in a failure report, which will be processed in a new iteration. Depending on the stage of the development process the error has been made, the effort reaches from small bug fixes to time consuming respecifications.

After correcting all errors, the application will be released by the test team.

List of artefacts

- test plan

  The test plan defines the proceeding during the test stage. It defines the scope, the strategy, the test methods and the tool support. The test plan is like the test cases defined previous to the actual tests [7].
- test cases

  In most cases the test cases are based on use cases and thus test a specific functionality. Test cases define each step of a test run and also its expected result. Often test cases are already used for development tests like JUnit tests.
- test report

  The test report contains the results of the test cases including all detailed informations about the test runs.
- failure report

  The failure report contains an exact description of all failed test cases including the results and errors them.

CMML for the test stage

To make successful testing possible, the test cases have to be directly related to the specifications from the analysis and design stage and to the application. The artefacts from the test stage might be related to artefacts from all previous development stages. This fact is represented by the test metamodel in figure 6.12.

[7] A short overview of the content of both can be found at `http://www.stellman-greene.com/aspm/content/view/39/41/` and [Reussner and Becker 2005]

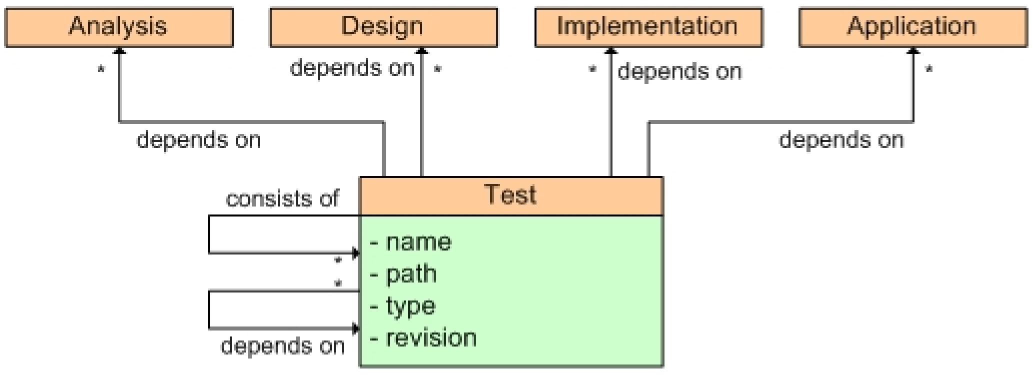


Figure 6.12: Metamodel for MDCM test

Because the implementation is tested against its specification, the test stage is connecting the finished implementation with the original analysed requirements. That's why testing often also discovers errors in the analysis stage. 6.13 shows an example of a test model.

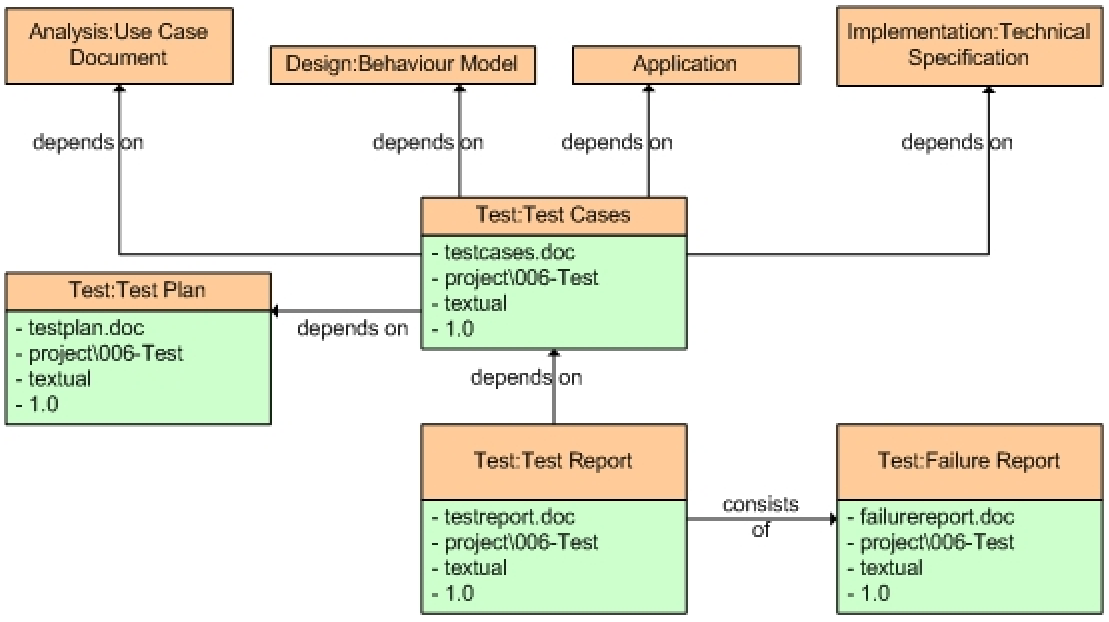


Figure 6.13: Test model based on Configuration Management Modeling Language

Test in MDSD

Because testing has its main focus on inspecting the manually created code, generating test cases is usually not possible. MDSM doesn't deal as much with the stage testing as it does with the other phases of the development process.

6.1.8 Deployment

After the test stage has been finished successfully, the deployment of the application on the production environment starts. Most parts regarding the deployment of the application had been planned in the architecture description during the design stage.

The deployment stage is responsible for managing the details of the deployment activities like the installation guideline and the integration plan.

List of artefacts

- installation guideline

  The installation guideline defines the necessary activities for the actual deployment. On the day of the deployment, the installation guideline will be executed step by step.
- integration plan

  Since the application to be deployed is usually not autonomous but rather depends on or uses different other systems, an integration plan has to be created. The integration plan defines how the new application is going to be inserted and integrated to the present system environment.
- data takeover plan

  Often new applications replace older applications. In this case there is already application data available which has to be taken over. The necessary steps are documented in the data takeover plan.
- user manual

  Unlike the other artefacts, the user manual is written directly for the customer. The user manual describes the functionality of the application and how to use it right.

After the necessary preparations had been completed, the application has to be deployed and if existent the application data has to be integrated. Depending on the size and importance of the application it is often required to start a trial run before actually make the application available to the customers.

After deployment the operation of the application starts. But this is not a topic of the development process and is thus not covered here. But there are many works and approaches like the service operations of ITIL [8] which cover this topic.

[8] http://www.itil.org/en/

CMML for the deployment stage

The artefacts of the deployment stage depend on the artefacts of the implementation stage. They describe the deployment activities as detailed as necessary. Figure 6.14 illustrates the CMML deployment metamodel.

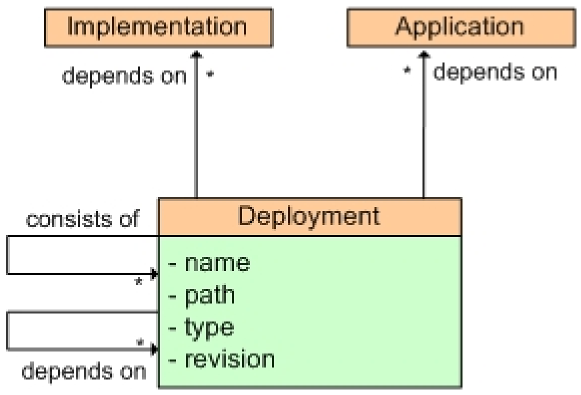


Figure 6.14: Metamodel for MDCM deployment

Figure 6.15 shows an example for a concrete deployment model.

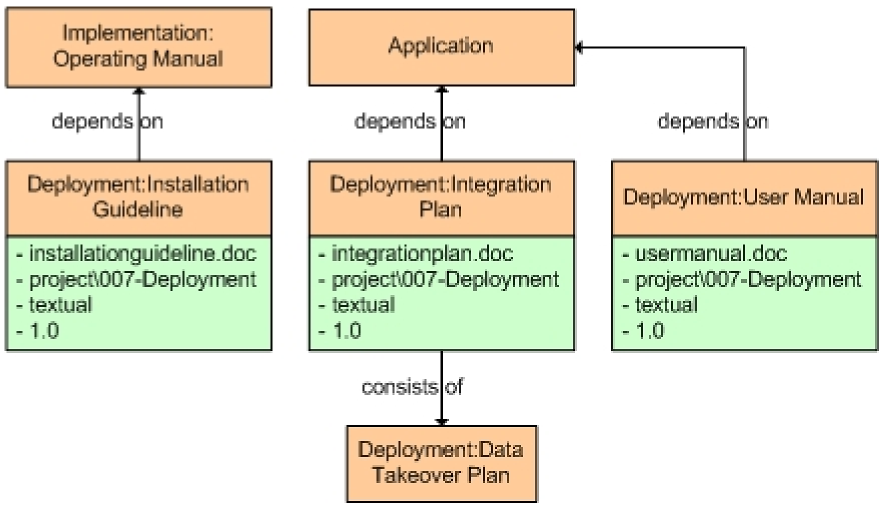


Figure 6.15: Deployment model based on Configuration Management Modeling Language

6.2 The Model Driven Configuration Editor in action

6.2.1 Model Driven Configuration Editor - How to create an editor for configurations

The MDCE will provide an easy to use graphical editor for configuration models and functions to map the models to a concrete configuration file. This section describes a procedure model, defines the needed system configuration to create the editor and provides an example needed later when a concrete editor is built.

General survey of the procedure model to create an editor

The MDCE procedure model helps building a configuration editor. Each step is explained here. Those steps are also and in greater detail described in the upcoming sections.

- Building the metamodel

  The first step is to build a metamodel, which describes the rules of the evolving editor. All elements and their connections are specified as classes and relations. The metamodel can be written in UML or Ecore (EMF). Using the MDCE approach, it is recommended to use Ecore in order to be able to apply GMF. An alternative solution is to generate the model from XSD.
- Generating the EMF model and the editor

  Based on the Ecore model or the XSD, it's easy to build a generator model which generates the EMF model and the editor. The EMF editor is a tree structured editor using the Eclipse workspace. It allows creating and changing files based on the Ecore model.
- Generating the GMF editor

  To make editing more comfortable, the next step builds a GMF editor. This editor can be built using an Eclipse cheat sheet (an interactive help producing the GMF files). However, the resulting editor is not yet proper suited for editing configurations.
- Adapt the GMF editor

  The existing GMF editor can now be extended and adapted to provide easier and individual editing. The MDCE procedure model offers some methods to map the EMF classes and connections to graphical GMF items. Using those methods makes the adaptation easier and quicker.
- Build the mapping

  The output of the files written in the GMF editor is already XML formatted. However, the format may not be the same as the desired result. Using transformation, the XML file can now be mapped into the right format.

- Creating the Plug-in

  The last step is to create the Eclipse Plug-in of the GMF editor including the editor itself and the mapping. The completed editor can then create, change and display configuration files. Editing will be more convenient and the display of the configuration is clear and concise.

Necessary Eclipse configuration

To be able to start with creating an MDCE, a proper working environment has to be set-up. The following components are used in this paper when developing the MDCE:

- Eclipse 3.2.1
- EMF-Eclipse Modeling Framework, the EMF-SDO-XSD SDK 2.2.1
- EMFT-Eclipse Modeling Framework Technology 1.0.1
- GEF - Graphical Editing Framework SDK 3.2.1
- GMF - Graphical Modeling Framework SDK 1.0.2
- openArchitectureWare 4.1.1 SDK
- JDOM 1.0 Library

The minimum set-up to run just oAW 4.1.1 is Eclipse 3.2 with Java JDK 5 and EMF 2.2.0. Although the listed configuration is recommended, more current versions can be installed and the described approach might still work [9].

The Naiad event server example

The following XML file will be used as an example configuration. Based on this file, the MDCE is created. This XML file is a simplified version of a configuration of an Naiad event server.

```
<?xml version="1.0" encoding="UTF-8"?>
<eventServer xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="eventserver.xsd">
  <correlationManager name="correlations">
    <tupleCorrelation identifier="sameDate"/>
  </correlationManager>
  <sessionManager name="sessions" correlationManager="correlations">
    <mergeManager name="mergeCount"/>
  </sessionManager>
</eventServer>
```

The XSD for this XML file looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified" attributeFormDefault="unqualified">
```

[9]A guide for installing EMF and GMF and the needed components can be found on the Eclipse project sites http://www.eclipse.org/modeling/emf/ and http://www.eclipse.org/gmf/

```
  <xs:element name="eventServer">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="correlationManager" minOccurs="0"
            maxOccurs="unbounded">
          <xs:complexType>
            <xs:element ref="tupleCorrelation"/>
            <xs:attribute name="name" type="xs:string" use="required"/>
          </xs:complexType>
        </xs:element>
        <xs:element name="sessionManager" minOccurs="0"
            maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="mergeManager">
                <xs:complexType>
                  <xs:attribute name="name" type="xs:string"
                      use="required"/>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
            <xs:attribute name="name" type="xs:string" use="required"/>
            <xs:attribute name="correlationManager" type="xs:string"
                use="required"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="tupleCorrelation">
    <xs:complexType>
      <xs:attribute name="identifier" type="xs:string" use="required"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Every MDCE example will work with these two files. The resulting editor will create configurations based on the XSD.

6.2.2 Create a model for a configuration

When mapping configurations to a model for using it in an MDCE, some rules have to be observed. The reason why this transformation is made, is to build an effective and powerful editor upon the model. The GMF editor will be based on the Ecore model and in order to get the best out of GMF, the Ecore model must have a specific structure and buildup. The next sections will describe the rules for converting the XSD elements to Ecore elements.

Elements, Attributes and Types

XSD elements will be represented by Ecore classes. One class for one element. Additionally to these classes, there is always a class called Diagram, which is the root class for all the other classes. The Diagram class represents the GMF drawing space and has therefore EReferences for all classes in the EMF. The attributes of the elements will be mapped to attributes of the Ecore class. The simple types of the XML Schema are mapped to Java types as far as they fit. Like an xs:string will be transformed to String, xs:int to int and so on. The text between the Element start-tag and end-tag has to be mapped as an attribute with the name text of the type String. Later when using the model-to-code transformation from the GMF model back to XML, the attribute text will be rebuilt. This is an additional effort when creating the model, because in an XSD, it doesn't have to be defined. But it's also an extra adjustable parameter.

Relations between Elements

Elements and subelements (children of the other element) will be transformed into two classes with a unidirectional relation. The relation is realised by having a class with the naming convention ClassnameChildClassnameRelation (for example, the ServerManagerSessionManagerRelation is connecting the ServerManager class with the child class SessionManager) and three attributes: name, source and target of the relation. The source class of the relation (in this case the Server class) has an EReference item pointing to the relation class (ServerManagerSessionManagerRelation). The cardinality of the relation is specified by the EReference item.

In case that there are Elements pointing on other elements, a relation with the naming convention ClassNamePointedClassnameConnection can be specified. Except for the different naming conventions, the procedure is the same as used when displaying regular relations. But when the model is transformed back into an XML file, the transformation process will handle both relations different.

Choice, Restriction and more

There are some XML Schema tags, which cannot be adapted that easily. Tags like Choice or Restriction don't have a counterpart in EMF. Those tags have to be modeled at a later stage of MDCE. Restrictions, the Choice tag and other additional constraints can be checked and modeled when mapping the GMF model back to XML.

The MDCE Ecore model for the Naiad event server XSD

This section shows the completed Ecore model based upon the Naiad event server XSD. Since this Ecore model is needed in the next chapters, it's recommended to make a new Eclipse project called EventServer, a folder named model and create the file EventServer.ecore in the model folder. There is a short excerpt of the XML representation of the model, displaying the SessionManager and afterwards there is a screenshot of the EMF model in Eclipse.

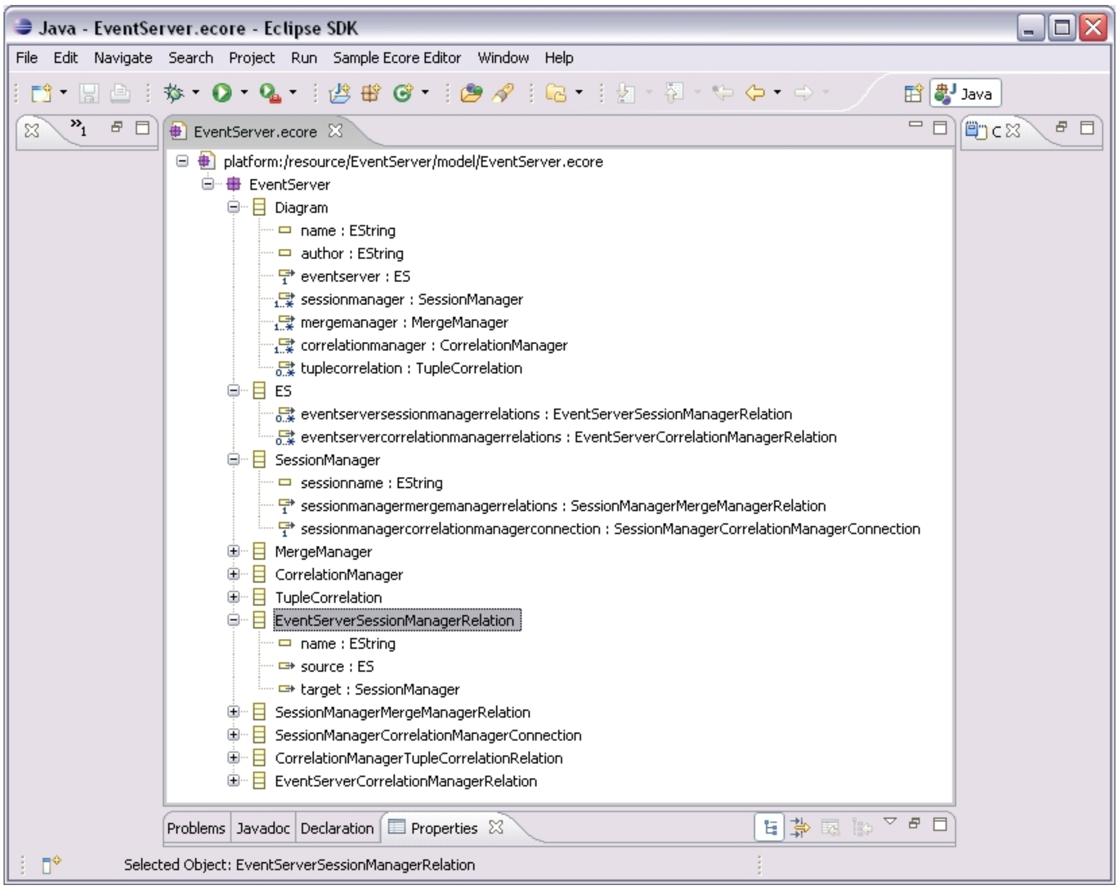
```
<eClassifiers xsi:type="ecore:EClass" name="SessionManager">
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="sessionname"
    eType="ecore:EDataType
    http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  <eStructuralFeatures xsi:type="ecore:EReference"
    name="sessionmanagermergemanagerrelations" lowerBound="1"
    eType="#//SessionManagerMergeManagerRelation"
    containment="true"/>
  <eStructuralFeatures xsi:type="ecore:EReference"
    name="sessionmanagercorrelationmanagerconnection" lowerBound="1"
    eType="#//SessionManagerCorrelationManagerConnection"
    containment="true"/>
</eClassifiers>
```

As can be seen, the SessionManager is an eClassifier by the type ecore:EClass. Attributes and References are both eStructuralFeatures with different types. The eType of the EReference element points to the class which handles the relation.

As can be seen in figure 6.16, there is a Diagram class containing links to all basic classes. The other classes have their attributes and references to the related classes. Classes ending with Relation are classes for linking two classes, one as source and one as the target of the relation. The complete EventServer.ecore file can be found in appendix EventServer.ecore.

After writing the ECore model, the genmodel is created. By selecting EMF Model under Eclipse Modeling Framework in the wizard for new files, the EMF genmodel is created by simply following the instruction. Right click on the root of the created EventServer.genmodel. Either select Create Model Code and Create Editor Code, which are the items needed by GMF, or select Create All in order to additionally create the test classes and the EMF editor.

This section has shown the theoretic fundamentals and a small example of building an EMF model for the MDCE. The example continues in the next section.


Java - EventServer.ecore - Eclipse SDK
File Edit Navigate Search Project Run Sample Ecore Editor Window Help
Java
EventServer.ecore
Problems Javadoc Declaration Properties

```
platform:/resource/EventServer/model/EventServer.ecore
  EventServer
    Diagram
      name : EString
      author : EString
      eventserver : ES
      sessionmanager : SessionManager
      mergemanager : MergeManager
      correlationmanager : CorrelationManager
      tuplecorrelation : TupleCorrelation
    ES
      eventserversessionmanagerrelations : EventServerSessionManagerRelation
      eventservercorrelationmanagerrelations : EventServerCorrelationManagerRelation
    SessionManager
      sessionname : EString
      sessionmanagermergemanagerrelations : SessionManagerMergeManagerRelation
      sessionmanagercorrelationmanagerconnection : SessionManagerCorrelationManagerConnection
    MergeManager
    CorrelationManager
    TupleCorrelation
    EventServerSessionManagerRelation
      name : EString
      source : ES
      target : SessionManager
    SessionManagerMergeManagerRelation
    SessionManagerCorrelationManagerConnection
    CorrelationManagerTupleCorrelationRelation
    EventServerCorrelationManagerRelation

Selected Object: EventServerSessionManagerRelation
```

Figure 6.16: The MDCE model of the Naiad event server
(A screenshot of the Eclipse EMF editor used for writing this model.)

6.2.3 Build a graphical editor for a model

Already an Ecore model of the Naiad event server configuration has been created. The next step is to create the GMF editor. To do this, the instructions of the cheat sheet have to be followed and the resulting models have to be modified in order to achieve a good working, effective editor.

The cheat sheet can be started in Eclipse by choosing the menu item Help/Cheat Sheets... . By selecting Graphical Modeling Framework/GMF Tutorial... at the Cheat Sheet wizard, the Cheat Sheet starts on the right side. Since a project and the domain model have been already created, the next step is Create a Graphical Definition.

Create a Graphical Definition

Left click on Click to Perform at the cheat sheet starts the wizard for creating the Graphical Definition. Select the EventServer/model folder. The file name is EventServer.gmfgraph. Then choose the EventServer.ecore as domain model. Select Diagram as Diagram Element and finish the wizard.

Let's take a look at the generated file/model EventServer.gmfgraph. In Eclipse, the file is shown by the EMF editor. The wizard has generated the nodes, connections and diagram labels. The next step is to modify the model to change its looks and behavior.

Presentation Changes

The standard view of MDCE for classes, connections and attributes is illustrated by the next image.

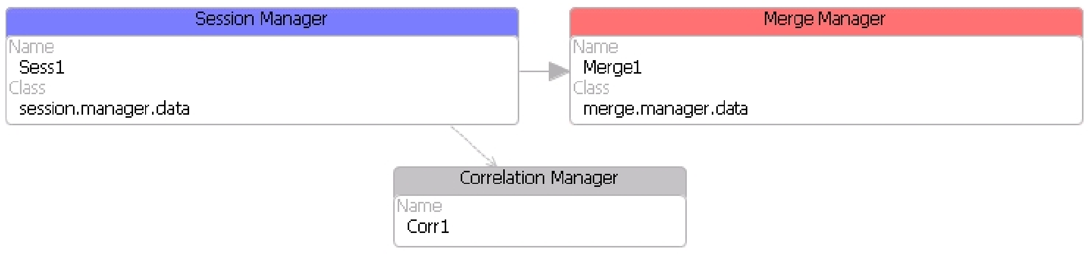
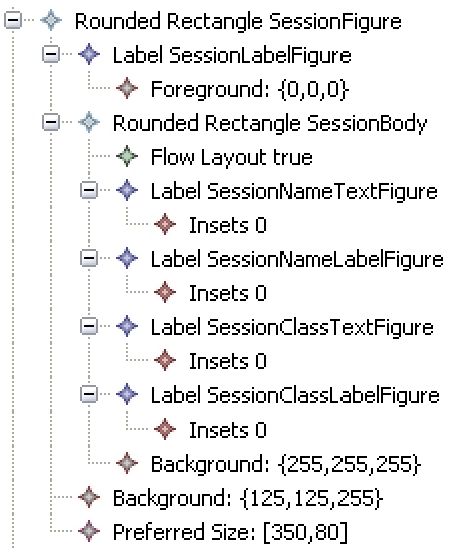


Figure 6.17: MDCE Figures
(This is the actual look/design of the MDCE components and connections. The different components can have different sizes, colors and attributes and there are different kinds of connections.)

To achieve this, the Figure Gallery elements have to be changed. The next picture and code example show how to model the figure to look this way. Both, classes and attributes are defined below.

A similar presentation has to be defined for every class including its attributes.

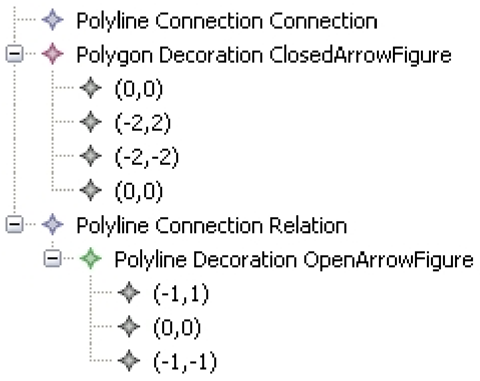


```
Rounded Rectangle SessionFigure
    Label SessionLabelFigure
        Foreground: {0,0,0}
    Rounded Rectangle SessionBody
        Flow Layout true
        Label SessionNameTextFigure
            Insets 0
        Label SessionNameLabelFigure
            Insets 0
        Label SessionClassTextFigure
            Insets 0
        Label SessionClassLabelFigure
            Insets 0
        Background: {255,255,255}
    Background: {125,125,255}
    Preferred Size: [350,80]
```

Figure 6.18: SessionManager Model

```
<figures xsi:type="gmfgraph:RoundedRectangle"
  referencingElements="Session"  name="SessionFigure">
  <children xsi:type="gmfgraph:Label"
     name="SessionLabelFigure" text="Session Manager">
     <foregroundColor xsi:type="gmfgraph:RGBColor"/>
  </children>
  <children xsi:type="gmfgraph:RoundedRectangle"
    name="SessionBody">
    <layout xsi:type="gmfgraph:FlowLayout"
      vertical="true" forceSingleLine="true"
      majorSpacing="0" minorSpacing="0"/>
    <children xsi:type="gmfgraph:Label"
      name="SessionNameTextFigure" text="Name ">
      <insets left="3"/>
    </children>
    <children xsi:type="gmfgraph:Label"
      referencingElements="SessionNameLabel"
      name="SessionNameLabelFigure" text="&lt;...>">
      <insets left="10"/>
    </children>
    <children xsi:type="gmfgraph:Label"
       name="SessionClassTextFigure" text="Class  ">
       <insets left="3"/>
    </children>
    <children xsi:type="gmfgraph:Label"
      referencingElements="SessionClassLabel"
      name="SessionClassLabelFigure" text="&lt;...>">
      <insets left="10"/>
    </children>
    <backgroundColor xsi:type="gmfgraph:RGBColor"
      red="255" green="255" blue="255"/>
  </children>
  <backgroundColor xsi:type="gmfgraph:RGBColor"
    red="125" green="125" blue="255"/>
  <preferredSize dx="350" dy="80"/>
</figures>
```

The next step is to change the presentation of the connection figures. The generated model has 5 Connection figures, but only 2 are necessary. Those 2 figures and their code are shown next.



```
Polyline Connection Connection
Polygon Decoration ClosedArrowFigure
    (0,0)
    (-2,2)
    (-2,-2)
    (0,0)
Polyline Connection Relation
    Polyline Decoration OpenArrowFigure
        (-1,1)
        (0,0)
        (-1,-1)
```

Figure 6.19: GMFConnections Model

```
<figures xsi:type="gmfgraph:PolylineConnection"
    referencingElements="AConnection" name="Connection"
    targetDecoration="ClosedArrowFigure"/>
  <figures xsi:type="gmfgraph:PolygonDecoration"
      name="ClosedArrowFigure">
    <template/>
    <template x="-2" y="2"/>
    <template x="-2" y="-2"/>
    <template/>
  </figures>
  <figures xsi:type="gmfgraph:PolylineConnection"
      referencingElements="ARelation"
      name="Relation" lineKind="LINE_DASH"
      targetDecoration="OpenArrowFigure">
    <children xsi:type="gmfgraph:PolylineDecoration"
        name="OpenArrowFigure">
      <template x="-1" y="1"/>
      <template/>
      <template x="-1" y="-1"/>
    </children>
  </figures>
</figures>
```

All changes concerning the Figure Gallery have been done. Now the rest of the Graphical Definition has to be modified. It has to be checked if every node, attribute and connection is related to the right figure. When this is done, the changes of the Graphical Definition are completed.

To check if the modifications are correct, see appendix EventServer.gmfgraph.

Create a Tooling Definition

Now select the Click to Perform Button of the Create a Tooling Definition part. Parent folder is still EventServer/model and the name of the Tooling Definition is EventServer.gmftool. Select the EventServer.ecore, Diagram as Diagram Element and finish the wizard. The next step is to modify the generated Tooling Definition.

First a Tool Group for the Relation/Connection elements is created. This group is called Links and the other one Nodes. Add one Creation Tool to Links called Connection and one called Relation. All other Creation Tools concerned with Connections/Relations can be deleted. Now the the description or arrangement of the Creation Tools may be changed. Every item of the toolbar has a standard GMF icon. To change those and add some self-provided icons, simply overwrite the image files located in the project folder EsEditor.edit/icons/full/obj16.

The complete and modified Tooling Definition can be found in appendix EventServer.gmftool.

Create a Mapping Definition

The Mapping Definition connects all GMF models. One more time, the wizard of the cheat sheet is used. Start the wizard, Parent folder is EventServer/model, the name of the file is EventServer.gmfmap. Now the other models have to be selected and loaded. After this, select Diagram as Diagram Root Element and finish the wizard.

Many elements of the generated model have to be changed. The nodes should be right the way they are, just their attributes (Label mappings) have to be connected to the fitting Diagram Labels. The Source and Target Feature of the Link Mappings have to be modified. Select Source for the Source Feature and Target for the Target Feature. Delete the Label Mappings of the Link Mappings since they will not be needed. The last modification is to check the Tool entries of all Node, Label and Link Mappings and correct them if necessary.

For the complete Mapping Definition, see appendix EventServer.gmfmapping.

Code Generation/Running the diagram

Right click on the EventServer.gmfmapping and select Create generator model. The created file EventServer.genmodel is responsible for code generation. This model has a lot of properties for generating code which can be modified or examined. Afterwards, right click the EventServer.gmfgen file and select Generate diagram code to generate the GMF editor.

To run the editor, start a new Eclipse Application (Run As.../Eclipse Application). Create a new Java Project. The editor can be found by creating a new diagram by selecting the EventServer Diagram at Examples in the Creation Wizard (New/Other...). Now its possible to paint the diagram and to use the toolbox to create new items and connect them. The whole GMF editor works as a Plug-in. To use the editor in Eclipse without starting a new Eclipse Application, the Plug-in Jar files have to be created using the export wizard provided by the Manifest file. Then copy the Jar files to the Plug-in folder of the Eclipse application.

The next and last step is to create a transformation from the model back to the configuration. The next chapter explains the different approaches.

6.2.4 Model-to-configuration transformation

This section describes the transformation of model to code. First openArchitectureWare and after that JDOM is used to do the transformation.

Transformation using openArchitectureWare

This section describes the model-to-configuration transformation of the EventServer example using the oAW framework.

The required configuration

It's required to have already created the EventServer GMF editor and EMF model. Additionally, create the Plug-Ins of these projects and copy them into the Eclipse Plug-in directory, so that the editor can be used.

Creating the oAW transformation

Start the development of the oAW transformation by creating a new Java project (with a separate source folder) named EventServer.oaw. Transform this project into a Plug-in project (Right click on the project, PDE Tools/Convert Projects to Plug-in Projects) and add the following dependencies to the manifest:

- all EventServer Plug-Ins
- all oAW libraries

Now oAW has to be enabled for this project. This can be done in the properties dialog of the project. The next step is to create the workflow for oAW. Therefore create a workflow.oaw and a workflow.properties file in the source directory. The property file has only these two lines:

```
modelFile=ES1.eventserver
srcGenPath=gen
```

The first line defines the input model, the second the output path. Now declare the property file in the workflow and add a parser to it. The parser reads the model file so that it can be used later for the transformation.

```
<workflow>
<property file='workflow.properties'/>
<component id="xmiParser"
    class="org.openarchitectureware.emf.XmiReader">
<modelFile value="${modelFile}"/>
<metaModelPackage value="EventServer.EventServerPackage"/>
<outputSlot value="model"/>
<firstElementOnly value="true"/>
</component>
</workflow>
```

Because the model is an XML file, the workflow component is an XmiReader. The tag metaModelPackage defines the location of the metamodel. In order to test the workflow, simply create a model named ES1.eventserver, add some items, connect them and then start the workflow (Right click on the workflow.oaw, Run As/oAW Workflow). If everthing is correct, the workflow should complete successfully.

The next step is to create a template for the transformation and add its execution to the workflow. Modify the workflow so that it first cleans the output directory and afterwards starts a template called mapping.xpt. After that, all left to do is write the transformation template. The following lines have to be added to the workflow.

```
<component id="generator" class="oaw.xpand2.Generator"
    skipOnErrors="true">
<metaModel class="oaw.type.emf.EmfMetaModel">
<metaModelPackage value="EventServer.EventServerPackage"/>
</metaModel>
<expand value="src::mapping::diagram FOR model"/>
<genPath value="${srcGenPath}/"/>
<srcPath value="${srcGenPath}/"/>
</component>
```

Use the Xpand2 generator of oAW. The input model and the first component of the model, which should be handled by the template, have to be defined. The last step is to write the template itself. Just a small excerpt of the template is explained here, but the entire template as well as the workflow and property file can be found in the appendix.

```
«DEFINE diagram FOR EventServer::Diagram-»
«FILE "Config.xml"-»
<?xml version="1.0" encoding="UTF-8"?>
«EXPAND es FOR eventserver-»
«ENDFILE-»
«ENDDEFINE»
«DEFINE es FOR EventServer::ES-»
  <eventserver>
    «EXPAND cm FOREACH eventservercorrelationmanagerconnection-»
    «EXPAND esr FOREACH eventserversessionmanagerconnection-»
  </eventserver>
«ENDDEFINE»
```

The template creates a file called Config.xml for the diagram of the model. Then it expands the eventserver, writes the tag (<eventserver>) and expands each connection. As can be seen in the appendix, each connection will be expanded and the attributes will be saved. All text which isn't between « and » is written into the file.

When the template has been completed, run the workflow and it should create the Config.xml file in the path defined by the property file. This file is the configuraton file. It's build-up and structure is exact the same as the EventServer.xml. It is now possible to easily and fast create configuration files for the EventServer. The MDCE is complete and ready to use.

Transformation using JDOM

Create a new Plug-in project named EventServer.jdom. Use the Plug-in with a popup menu template to create the new project. The name filter for the menu is *.eventserver. Write transform... into the submenu name field and choose create Config.xml as action label. Then alter the manifest and add all EventServer Plug-Ins to the dependencies and import all org.jdom packages.

Next step is to develop the transformation. The easiest way to create it is to extend the already existent NewAction.java file. The first step is to load the model into the JDOM parser. Hence create an IFile Object, open the selected file, create a JDOM Document of the file and an additional empty JDOM Document for the transformation result. The methods written therefore are shown in the listing below.

```
public void selectionChanged(IAction action,
    ISelection selection) {
  file=(IFile)((IStructuredSelection)selection).getFirstElement();
}

public Document buildDomDocument() throws Exception {
  SAXBuilder builder = new SAXBuilder();
  Document D = builder.build(file.getLocation().toFile()
    .getAbsolutePath());
  return D;
}

public Document createEmptyDomDocument() throws Exception {
  Document doc=new Document();
  return doc;
}
```

The first method determines the file name, the second opens the file, builds a JDOM Document with the components of the file and the third method simply creates an empty JDOM Document.

The core of this class is the recursive transformation method addElement. It is called for each child of the root element, scans the JDOM model and adds a children to the new model for each found relation. Then the method calls itself, this time with the child as a parameter. addElement uses two helper methods, getChildrenRef which determines the child name of a connection and createElement to clone an element.

```
public void addElement(Element root,Element addEl,
    Element deep,String ref) throws Exception {
  Element Childref=getChildrenRef(root,deep,ref);
  Element item=createElement(Childref);
  List childs=Childref.getChildren();
  for(int j=0;j<childs.size();j++) {
    String targetVal=((Element)childs.get(j))
      .getAttributeValue("target");
    String targetName=((Element)childs.get(j)).getName();
    if(targetName.endsWith("relation")) {
      Element att=getChildrenRef(root,item,targetVal);
      item.setAttribute(att.getName(),att
         .getAttributeValue("name"));
    }else {
      if(targetVal !=null) {
        addElement(root,item,Childref,targetVal);
      }
    }
  }addEl.addContent(item);
}
```

Now that the transformation has been developed, a method to save the new model has to be written. This method saves the JDOM Document and uses the path of the original file. The method is listed below.

```
public void saveDocument(Document E) throws Exception {
String fname=file.getLocation().toString();
int ind=fname.lastIndexOf("/");
fname=fname.substring(0, ind+1);
File f=new File(fname+"output.xml");
FileOutputStream fo=new FileOutputStream(f);
    XMLOutputter out = new XMLOutputter(Format.getPrettyFormat());
    out.output(E,fo);
}
```

The complete NewAction.java can be found in the appendix . After finishing the modifications, simply create a Plug-in of the project and copy it to the Plug-in folder of Eclipse. Now it is possible to create a configuration file by simply choosing the popup menu transform.../create Config.xml of the EventServer model.

7 Example Scenarios - Model Driven Configuration Management applied

This chapter shows concrete uses of the Model Driven Configuration Management (MDCM) approach. These two scenarios show only a small part of MDCM, but they provide a lot of information about how to apply MDCM and how to use the Model Driven Configuration Editor (MDCE) to create files based on models.

Section 7.1 Modeling a simple configuration file deals with an important part of MDCM. The creation, change and representation of models based on XML configuration files. This section delivers the knowledge which had provided the foundation for MDCM and the MDCE and is an essential part of it. Section 7.1 doesn't use a concrete MDCE, but it uses the different components the MDCE is based on and therfore provides necessary informations how the MDCE operates.

Section 7.2 The flightplan example described how to apply the MDCM process to a small real world example. All important parts of MDCM including the procedure model, the Configuration Management Modeling Language (CMML) and the MDCE are explained.

7.1 Modeling a simple configuration file

This chapter describes how to create models for XML configurations. A simple configuration example is mapped into both, a Unified Modeling Language (UML) model and an Eclipse Modeling Framework (EMF) model. This section also shows how to map an XSD into a model.

Both modeling languages, UML and EMF, have some standard objects for defining the structure and composition of the configuration. For describing the configuration with UML, it's best to use a UML class diagram. Both languages have the following components for defining the configuration:

- Class: Defines the elements of the configuration. For example, a network diagram might have the classes workstation, router and server.
- Attributes: Classes can have attributes for further describing them. A workstation might have the attributes name, cpu speed and hard disc space.
- Relations: Classes can be connected by relations. In this example, the workstations and servers are connected and the router is connected to the server and the internet.

7.1.1 Metamodels for configurations

It may be necessary to create a metamodel for the configuration to exactly specify the model. In the network example, the metamodel can have the following items:

- I/O Device: Workstations and servers are components of this item. Several properties can be defined for the workstation and server items.
- Network Device: The router is a component of this metamodel item.
- Network Connection: Every connection defined in the model is one of the type network connection.

There are three possible starting states for building the metamodel:

- An XML Schema already exists for the configuration

  The Schema defines the structure of the configuration. In this case the metamodel can be converted from the Schema. There is already work available about this subject (see section 3.4.3 Creating a model out of an XML Schema).
- There are rules available, describing the configuration

  A metamodel based on that rules has to be written. No procedure can be defined for this task, since the requirements and problems of creating the metamodel differ from configuration to configuration.
- No rules defined yet

  Starting from the knowledge about the needed configuration, rules have to be described. These rules include the objects of the configuration and their interaction. Based on that rules, the metamodel is written.

7.1.2 The Naiad example

The following XML file will be used as an example configuration. Based on this file, the next sections create an EMF and UML model. This XML file is a simplified version of a configuration of a Naiad event processor (for further information see chapter 7.2 The flightplan example

```
<?xml version="1.0" encoding="UTF-8"?>
<eventServer xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="eventserver.xsd">
  <correlationManager name="correlations">
    <tupleCorrelation identifier="sameDate"/>
  </correlationManager>
  <sessionManager name="sessions" correlationManager="correlations">
    <mergeManager name="mergeCount"/>
  </sessionManager>
</eventServer>
```

The XSD for this XML file looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified" attributeFormDefault="unqualified">
  <xs:element name="eventServer">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="correlationManager" minOccurs="0"
            maxOccurs="unbounded">
          <xs:complexType>
            <xs:element ref="tupleCorrelation"/>
            <xs:attribute name="name" type="xs:string" use="required"/>
          </xs:complexType>
        </xs:element>
        <xs:element name="sessionManager" minOccurs="0"
            maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="mergeManager">
                <xs:complexType>
                  <xs:attribute name="name" type="xs:string"
                      use="required"/>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
            <xs:attribute name="name" type="xs:string" use="required"/>
            <xs:attribute name="correlationManager" type="xs:string"
                use="required"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="tupleCorrelation">
    <xs:complexType>
      <xs:attribute name="identifier" type="xs:string" use="required"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

The next section starts now with writing an EMF model for this configuration.

7.1.3 Modeling with Eclipse Modeling Framework

At first it must be determined if a new metamodel has to be created or if the already existing one can be used. The standard metamodel has a close relation to Java. The modeling elements are all well known to Java developers. The following list shows the most important items provided by the basic metamodel of EMF.

- **EPackage** simply groups elements into packages. When creating Java code out of EMF, those packages are transformed into Java packages.
- **EClass** is the main modeling element of EMF. Remembering our network example, workstations, servers and routers would be modeled using the EClass. EClass elements are transformed into Java classes.
- **EAttribute** elements define attributes for an EClass. They are like Java attributes.
- **EOperation** defines an operation for an EClass. They will be transformed into Java methods.
- **EReference** is used to model relations between EClass elements.
- **EAnnotation** is simply a note for describing a modeled item.

These modeling elements are sufficient for modeling our Naiad example. But nevertheless an Ecore model is written for demonstration purpose.

Writing an Ecore model

The Ecore model will be built as illustrated in the following UML diagram.

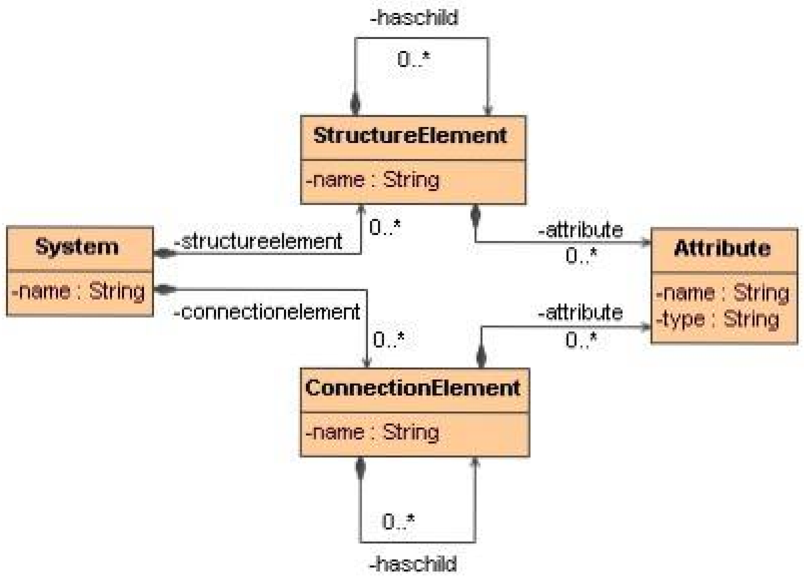
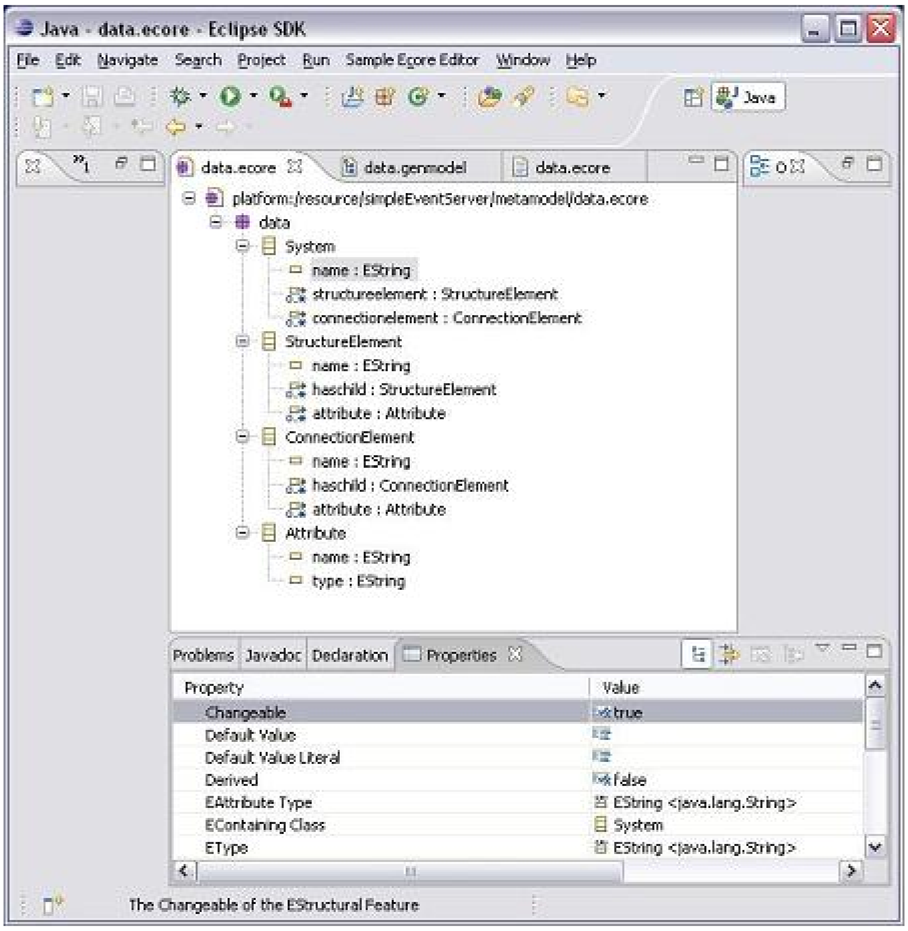


Figure 7.1: The Ecore metamodel
(This metamodel defines the components of an Ecore model. It is also the metametamodel of an EMF model.)

Based on this metamodel, a model for the simpleEventServer configuration can be written. The metamodel contains a System class for the main model object, a StructureElement class which includes the managers of the configuration and a ConnectionElement class for correlations. Furthermore, both classes can have attributes to describe them more exactly.

The metamodel can be written using the Eclipse EMF editor. The elements are modeled using EClass, the attributes using EAttribute and the relations are modeled as EReference. Here is a screenshot of the EMF editor of Eclipse.


Java - data.ecore - Eclipse SDK
File Edit Navigate Search Project Run Sample Ecore Editor Window Help
Java
data.ecore
data.genmodel
data.ecore
Problems Javadoc Declaration Properties

```
platform:/resource/simpleEventServer/metamodel/data.ecore
  data
    System
      name : EString
      structureelement : StructureElement
      connectionelement : ConnectionElement
    StructureElement
      name : EString
      haschild : StructureElement
      attribute : Attribute
    ConnectionElement
      name : EString
      haschild : ConnectionElement
      attribute : Attribute
    Attribute
      name : EString
      type : EString
```

Property	Value
Changeable	true
Default Value	
Default Value Literal	
Derived	false
EAttribute Type	EString <java.lang.String>
EContaining Class	System
EType	EString <java.lang.String>

The Changeable of the EStructural Feature

Figure 7.2: Standard Eclipse editor for the Eclipse Modeling Framework
(The main window is a tree view of the different EMF components. The window below is used to display the different attributes of the item selected in the main window.)

The Ecore model is saved as an XML file. The following code extract of the Naiad Ecore model describes the System class.

```
<eClassifiers xsi:type="ecore:EClass" name="System">
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"
   eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  <eStructuralFeatures xsi:type="ecore:EReference"
   name="structureelement" upperBound="-1"
   eType="#//StructureElement" defaultValueLiteral="" containment="true"/>
  <eStructuralFeatures xsi:type="ecore:EReference"
   name="connectionelement" upperBound="-1"
   eType="#//ConnectionElement" containment="true"/>
</eClassifiers>
```

The code sample above illustrates how EMF saves an Ecore model. There are EAttribute, EReference and EClass tags. The xsi:type describes the type of the Ecore element, the eType tag specifies the type of the element, like an EReference is linked to the target element and an EAttribute is linked to an Ecore data element.

Having the completed metamodel, the generator model can be created. This is a simple task when using Eclipse since it is generated. Further information on this topic can be found at the EMF example [VOELTER] by openArchitectureWare. Using the generator model the Java classes and an editor for the metamodel can be created. The easiest way to run this editor is to start a new Eclipse workbench with a new project. The editor for the model can be found at New/Other... in the folder Example EMF Model Creation Wizards.

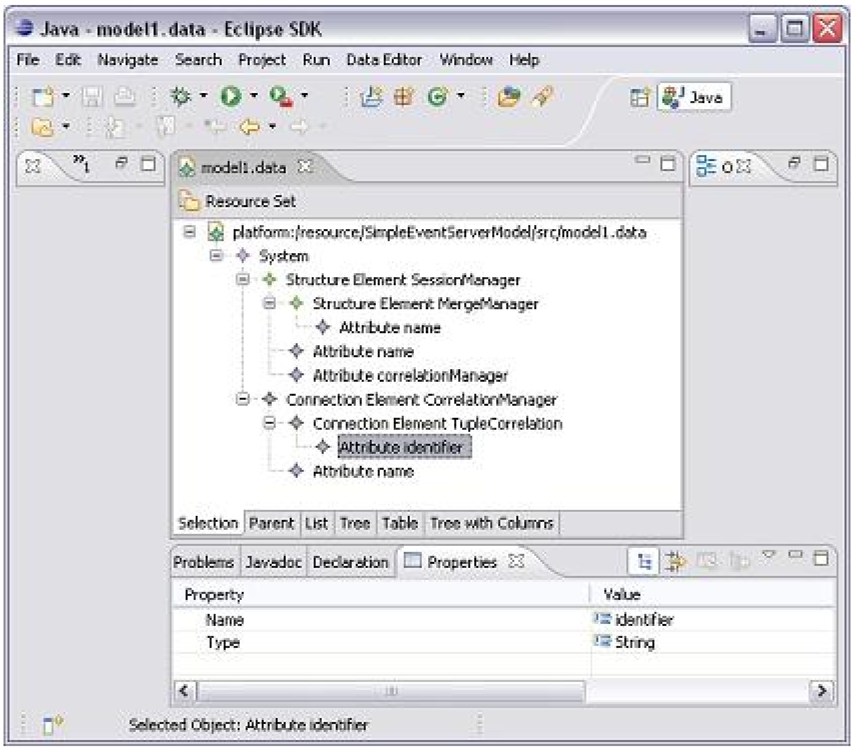
Writing the EMF model

When using the editor, most parts of writing the EMF model are similar to writing the Ecore model. But instead of having the data types EClass, EAttribute and so on, the elements are now previously defined in the Ecore model. So a StructureElement is created with the name SessionManager. The result is a model, based on the Ecore model, representing the XSD of the Naiad. The model is written in XML and the code below shows how the file looks like.

```
<?xml version="1.0" encoding="UTF-8"?>
<data:System xmi:version="2.0"
 xmlns:xmi="http://www.omg.org/XMI" xmlns:data="http://data">
  <structureelement name="SessionManager">
    <haschild name="MergeManager">
      <attribute name="name" type="String"/>
    </haschild>
    <attribute name="name" type="String"/>
    <attribute name="correlationManager" type="String"/>
  </structureelement>
  <connectionelement name="CorrelationManager">
    <haschild name="TupleCorrelation">
      <attribute name="identifier" type="String"/>
    </haschild>
    <attribute name="name" type="String"/>
  </connectionelement>
</data:System>
```

The model file is simply an XML file, where the tags represent the elements. Based on this model file, Model Driven Architecture (MDA) or Model Driven Software Development (MDSD) tools can be used to generate Java Code, Documentation, database tables and even more. An openArchitectureWare example demonstrates this [VOELTER].

Figure 7.3 displays the model in the EMF editor.


Java - model1.data - Eclipse SDK
File Edit Navigate Search Project Run Data Editor Window Help
Java
model1.data
Selection Parent List Tree Table Tree with Columns
Problems Javadoc Declaration Properties

```
Resource Set
⊟ platform:/resource/SimpleEventServerModel/src/model1.data
  ⊟ System
    ⊟ Structure Element SessionManager
      ⊟ Structure Element MergeManager
          Attribute name
        Attribute name
        Attribute correlationManager
    ⊟ Connection Element CorrelationManager
      ⊟ Connection Element TupleCorrelation
          Attribute identifier
        Attribute name
```

Property	Value
Name	identifier
Type	String

Selected Object: Attribute identifier

Figure 7.3: The Naiad model
(This EMF model provides the needed Elements, Connections and Attributes to define a model defining a configuration of an Naiad event server)

Another approach of using EMF

This was the usual way of modeling a system. But there is another possibility to use the Ecore and EMF models. In the examples above, Ecore models aspects of the UML layer 2, the metamodel layer. But it is possible, to define layer 1, the model layer, with an Ecore model. When modeling this way, the EMF model based on the metamodel declares items of layer 0, the data layer. The EMF editor can than be used directly to model data elements. This approach is extended by the MDCE when writing a GMF editor for the data layer model. This topic is covered by the section 6.2.2 Create a model for a configuration.

7.1.4 Modeling with the Unified Modeling Language

Modeling with UML is similar to modeling with EMF. This is not a great surprise, since EMF implements many features of UML. At the start, the creation of a metamodel has to be considered. In UML, MOF is used for creating a metamodel. Since UML is not that important for the MDCE than EMF, this work won't show an example of an MOF. But it is not much different from the creation of an Ecore metamodel.

When, for example using MagicDraw (one of the most popular UML tools), there is a list of all MOF items, which can be easily changed in order to create an own MOF definition. This definition can then be used to create a UML diagram based on the modified metamodel.

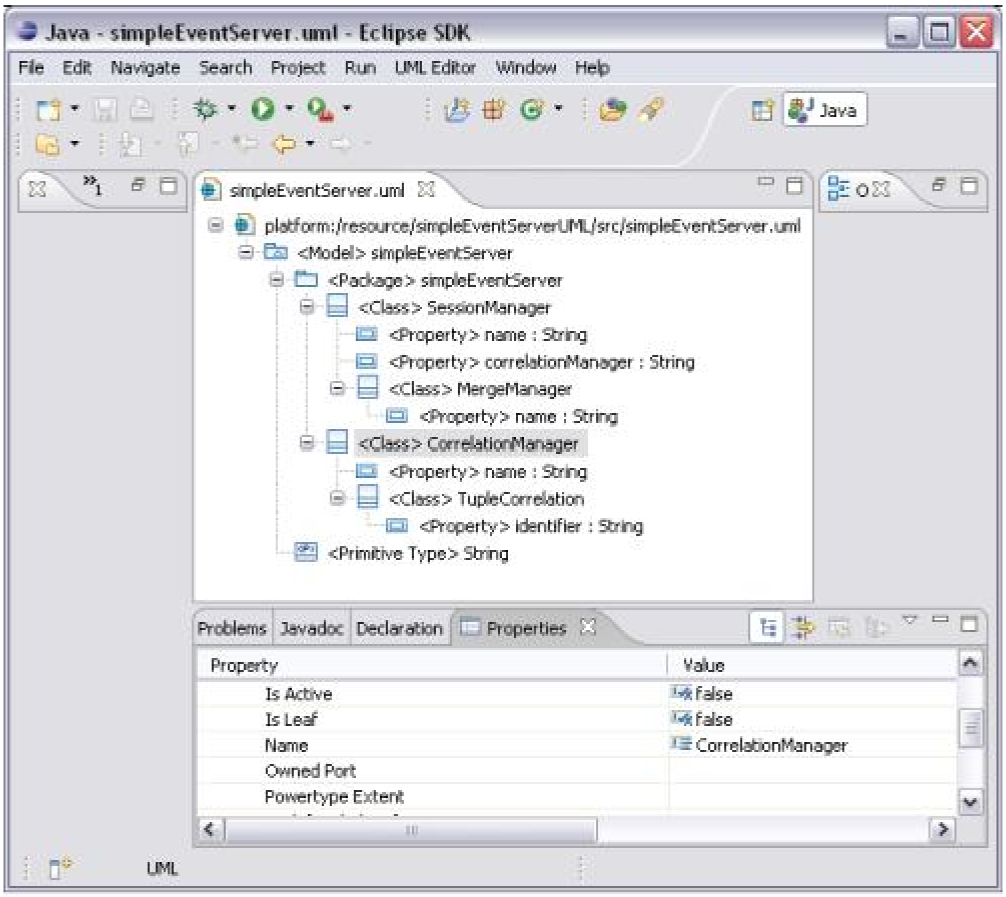
Next a UML model is created with UML2, an Eclipse Plug-in which defines all UML elements with Ecore. UML2 is used because Eclipse has already been used for EMF and it is free of charge.

Create a model with Eclipse UML2

Version 2.0.2 of UML2 and the MDT Tools is used for writing the UML model. First, start Eclipse, create a new Java Project and a UML model with Model as model object. A UML model can now be created. The editor is the same than when writing an EMF model. The following elements are used for modeling the Naiad example.

- **Model** is the root element, defining the name and other properties of the model.
- **Packaged Element Package** is the same element as EPackage. It is used to generate a package for the underlying elements.
- **Packaged Element Class** defines the model element class. This element will be used for defining the different configuration items.
- **Owned Attribute Property** is used to declare an attribute of a class, for example name or type or whatever is needed.
- **Packaged Element Primitive Type** defines a data type which is used for declaring the type of an attribute, like String or Integer.
- **Packaged Element Association** defines the associations between the created classes.

There are two ways to model the Naiad file. One is to define XML tags, which are children of other tags as a class with a nested class. This is shown by figure 7.4 on the next page.


Java - simpleEventServer.uml - Eclipse SDK
File Edit Navigate Search Project Run UML Editor Window Help
Java
simpleEventServer.uml
Problems Javadoc Declaration Properties
UML

```
platform:/resource/simpleEventServerUML/src/simpleEventServer.uml
  <Model> simpleEventServer
    <Package> simpleEventServer
      <Class> SessionManager
        <Property> name : String
        <Property> correlationManager : String
        <Class> MergeManager
          <Property> name : String
      <Class> CorrelationManager
        <Property> name : String
        <Class> TupleCorrelation
          <Property> identifier : String
    <Primitive Type> String
```

Property	Value
Is Active	false
Is Leaf	false
Name	CorrelationManager
Owned Port	
Powertype Extent	

Figure 7.4: An UML2 model
(The UML2 editor uses the EMF Plug-In as a foundation to model UML class diagrams. The editor provides the same tree view as the ECore/EMF editor does.)

The second way is to define the children as a relation of one class to another. This way is used by the MDCE and more information about it can be found in section 6.2.2 Create a model for a configuration. The UML model which has been created can be used (like the EMF model) as input for MDA or MDSD in order to generate code. It is often used also for documentation purpose.

7.2 The flightplan example

7.2.1 Introduction

This chapter illustrates the use of Model Driven Configuration Management (MDCM) and of the Model Driven Configuration Editor (MDCE) by modeling a small real world application. A small software and development model will be written, editors for supporting, creating and modifying the central crucial configuration files will be built and the use and the limits of this approach will be demonstrated. This chapter can also be seen as a tutorial explaining the basic features of MDCM by modeling this small application. The flightplan example is a small example which shows the core functions and usability of the IFS CEP.

CEP stands for correlated event processor. It is a concept for creating a common platform for event processing and correlation for various purposes [BERGER 2006]. CEP uses the implementation of the Mule framework [1], is written in Java 5 and uses Maven as build tool. CEP is a project of the IFS department of the technical university of Vienna. About 10 people are currently (June 2007) working on this project.

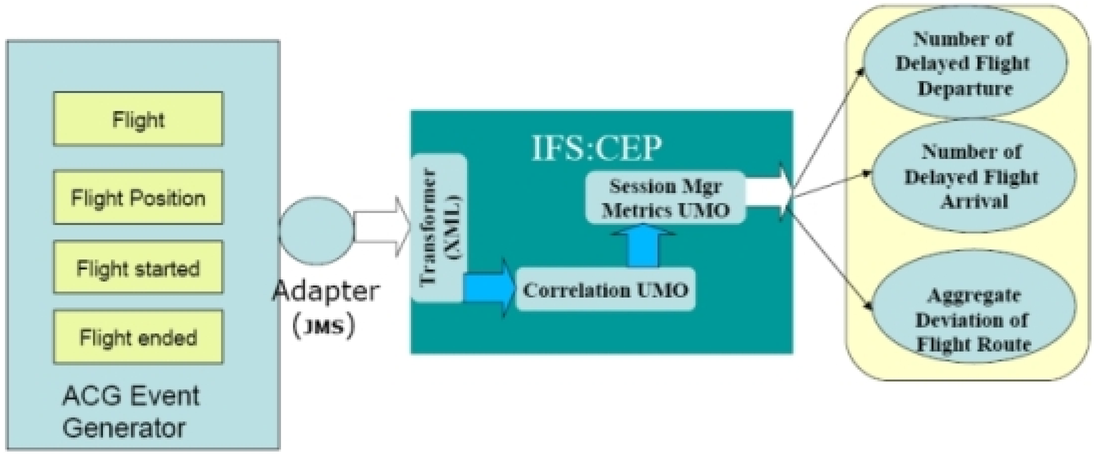


Figure 7.5: Flightplan - CEP diagram from [SAURER 2006]
(Describing the relation between the flightplan example and the correlated event processor)

The basic set of services provided by CEP are [BERGER 2006]:

- Transparent event processing
- Correlation/Session management
- Metrics
- Indexing
- Persistence

[1] See http://mule.codehaus.org/display/MULE/Home

The flightplan example uses the event processing of CEP for defining and measuring flights, their route and properties. Figure 7.5 shows the connection between the flightplan and the CEP. Since the flightplan example is still enhanced and changed, the diagrams and models will not reflect the flightplan configuration when its finished. They represent just a snapshot (18.06.07) of the application. This is not really a problem because this chapter should describe the use of MDCM and not necessarily be a documentation of the flightplan example.

7.2.2 Technologies and frameworks used by CEP/flightplan

Since CEP and the flightplan example are developed by one team, both applications use the same development environment. The flightplan example and the CEP Core are both written in Java 5. The event processing is done using the Mule framework. Testing is supported by JUnit tests. The building and deployment of the core and of the flightplan is handled by Maven. Maven also generates the online documentation and Java API.

Trac serves as project planning and management tool. Milestones and tasks are managed by Trac. Configuration management including change management, history and versioning is provided by Subversion.

In summary both projects are developed using the following technologies:

- Java 5
- Mule
- JUnit
- Maven
- Subversion for configuration management (CM)
- Trac for project management (PM)

Model Driven Configuration Management applied to the flightplan example

The MDCM approach will be used to write a development model of the flightplan example. All necessary configuration items of the example application are defined in this model. The development model provides a clear view of all classes and packages and their connections. But since there is no Subversion support for MDCM, the model can not be used as an input for configuration management. The software configuration is also defined as a model which again has the main function to offer a good view on the used software Configuration Items (CI).

The flightplan example uses one file for the configuration of mule and one for the configuration of the Naiad event server. Both files are written in XML. An editor for writing and modifying these two files will also be provided as a part of MDCE.

7.2.3 Model Driven Configuration Management models

In this chapter the software and development configuration of the flightplan example are described using MDCM. The models for each of these two configurations will be designed and created next. The first model describes the necessary software configuration of the flightplan example.

Software model

The software configuration model for the flightplan example describes the development environment. It contains the Java version and the necessary Plug-Ins and libraries. This configuration model describes which software has to be available in order to be able to develop this project. Figure 7.6 displays the model.

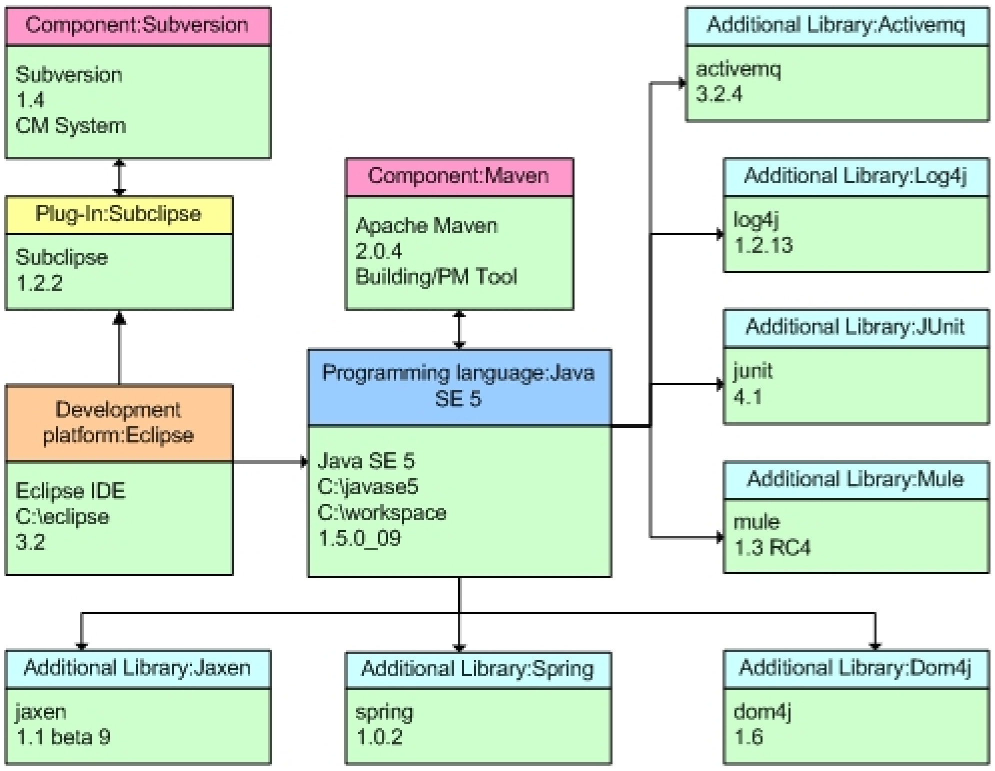


Figure 7.6: Software model for CEP/flightplan example
(Describing the different components and configuration elements relevant for the target application)

This model doesn't include software components like the operating system and the project management tool Trac. The software model has the main purpose to provide a clear view of the important CIs. The author of the model has to decide which CIs are added to it.

Development model

The development model is the most important part of MDCM. It describes the structure and dependencies of an application under development. Since even a small application has many packages and classes, including all of them into one model is not possible. The flightplan project will be described by a project level and by a package level model.

The model shown in figure 7.7 describes the flightplan project by showing all its packages and connections to other projects.

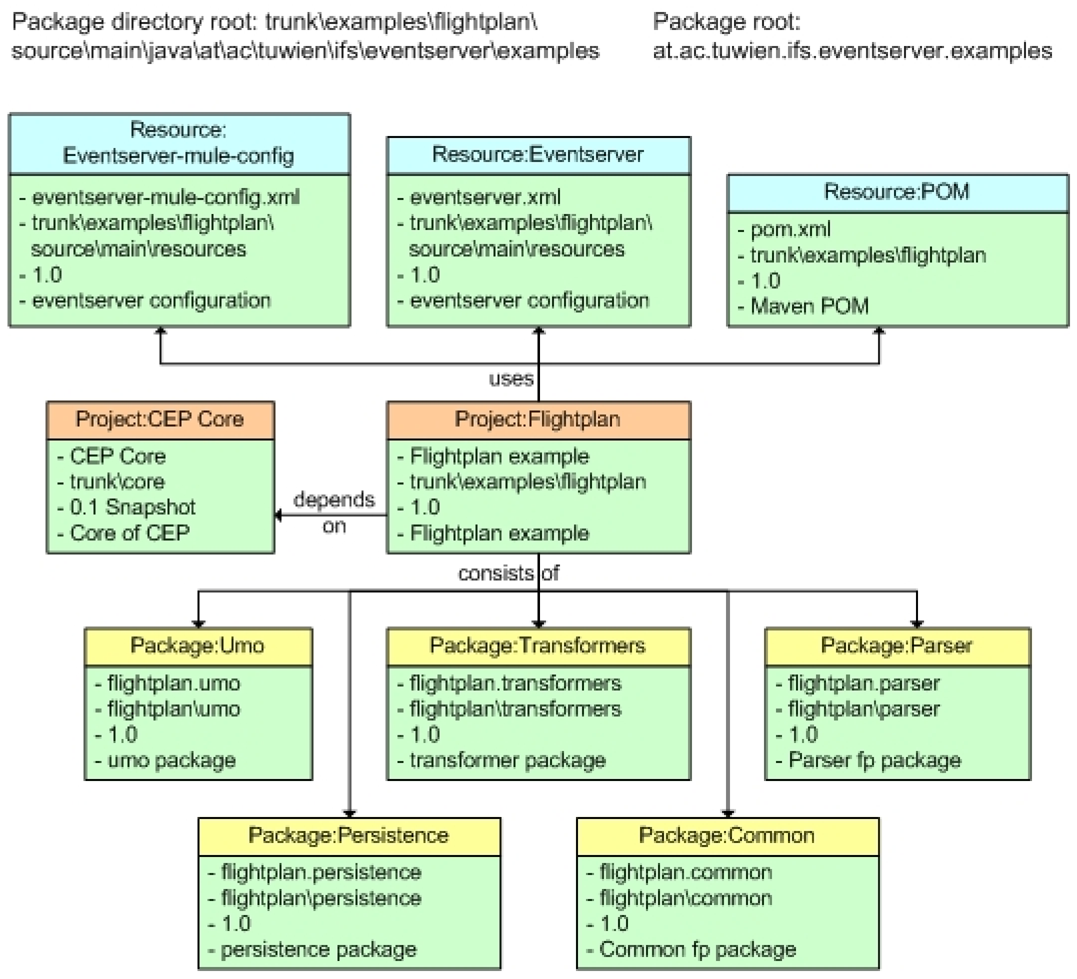


Figure 7.7: Development model of the project view for the flightplan example

The important packages persistence and common are also described by a class view diagram. All classes and their dependencies are modeled here. The class view displays the exact build up of a package. Figure 7.8 illustrates the class view diagram.

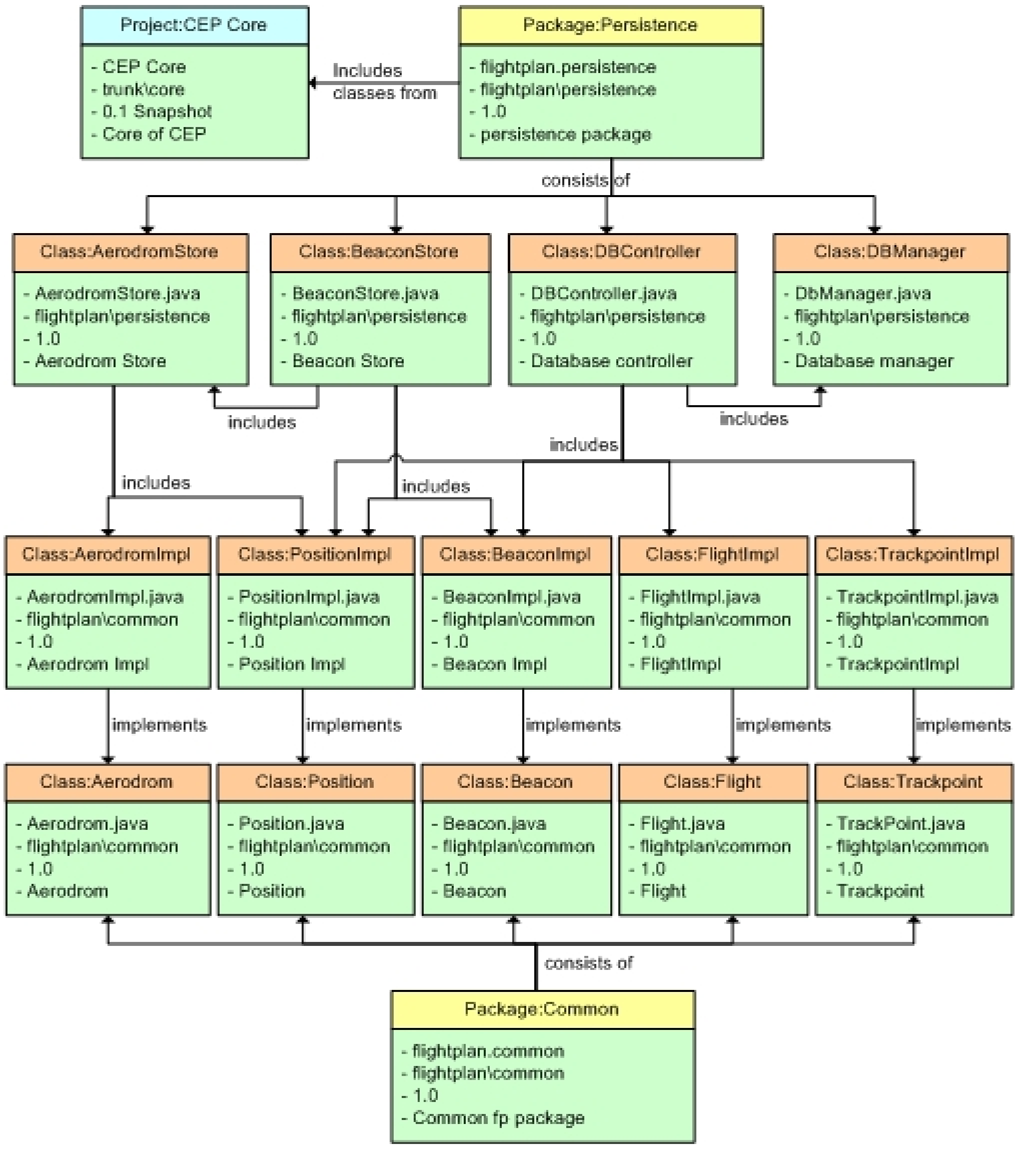


Figure 7.8: Development model of the package view of the common/persistence package

Including the models to Configuration Management

The models of MDCM, especially the development model, offer a real good foundation for configuration management (CM). The important CIs have been already described in those models and therefore the MDCM models would be the perfect input for a CM tool. However, most tools do not support the import of CIs. Subversion, which is used as CM tool in the flightplan project, doesn't offer a way to import the MDCM models.

The MDCM models can also be used as an input for change management. All relations and dependencies between the CIs can be found in those models. Dependencies can be defined at different levels. The more specific the model becomes, the better the dependencies are defined and the easier it gets to perform change control.

7.2.4 Building and using a prototype for development models

This work has described the theoretic aspects of MDCM and each part and technology of it. Although there are a lot of examples, not one of them shows a complete MDCM workflow. Therefore, this chapter explains how to create and use MDCE and the development model to create input for a CM suite as well as to generate Java classes. Each single step of the whole development process will be covered. The result will be a prototype of an editor for development models. This prototype shows the potential and usage of the development model. Section 7.2.4 Using the prototype deals with describing these potentials. First an XML Schema for the development model has to be defined.

XML files as input for configuration management tools

To create an XML Schema which is able to represent every aspect of the MDCM development model would be very difficult and time-consuming. Since this example only creates a prototype, the XML Schema only defines the package view including the following elements:

- project
- package
- class
- includes relation between classes
- implements relation between classes

Class elements will be saved as child elements of packages which will be children of a project. These elements all relate to figure 6.9, which describes the metamodel of the development model. The XSD will not exactly represent this metamodel since its main purpose is to define rules to describe the model of figure 7.8. The following listing shows an XSD to meet these definitions.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:element name="Class">
    <xs:complexType>
      <xs:choice>
```

```
        <xs:element ref="IncludesClass" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="ImplementsClass" minOccurs="0"/>
      </xs:choice>
      <xs:attribute name="name" type="xs:string" use="required"/>
      <xs:attribute name="version" type="xs:string" use="required"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="ImplementsClass">
    <xs:complexType>
      <xs:attribute name="name" type="xs:string" use="required"/>
      <xs:attribute name="package" type="xs:string" use="required"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="IncludesClass">
    <xs:complexType>
      <xs:attribute name="name" type="xs:string" use="required"/>
      <xs:attribute name="package" type="xs:string" use="required"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="Package">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Class" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="name" type="xs:string" use="required"/>
      <xs:attribute name="version" type="xs:string" use="required"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="Project">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Package" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="name" type="xs:string" use="required"/>
      <xs:attribute name="path" type="xs:string" use="required"/>
      <xs:attribute name="version" type="xs:string" use="required"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

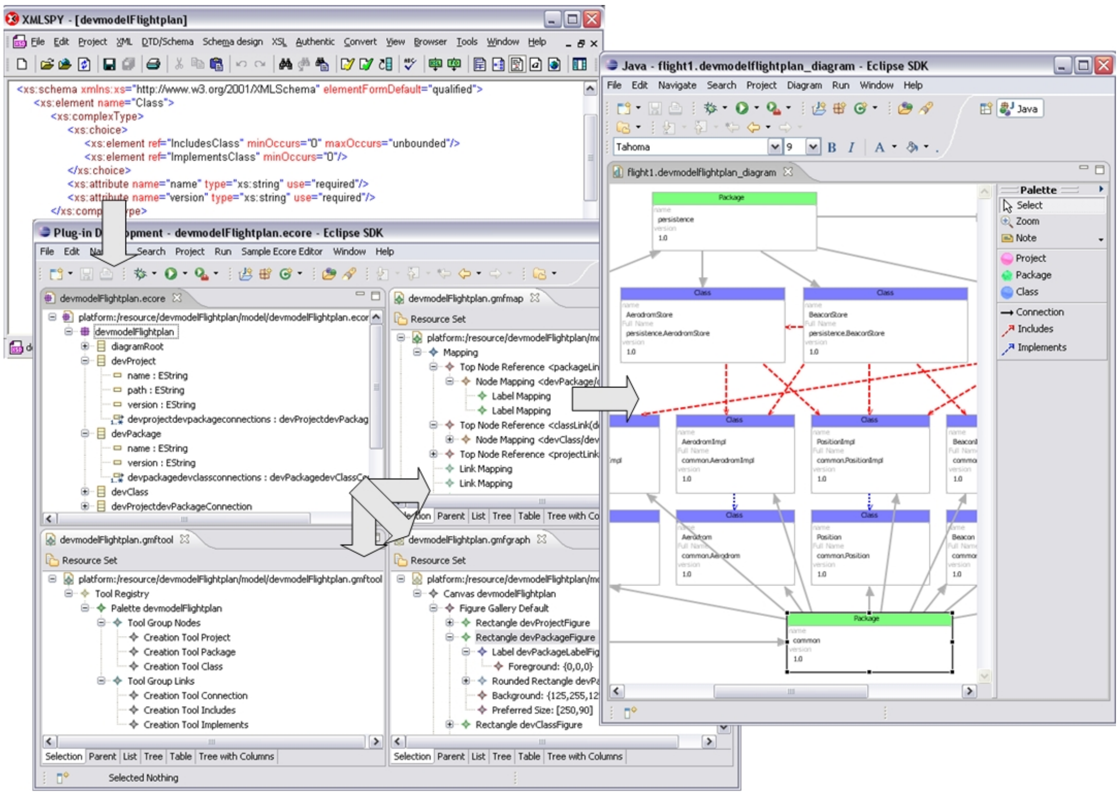
In the next section an editor based on this XSD will be built.

Building an Model Driven Configuration Editor for development models

Firstan Ecore metamodel based on the XSD is created. Therefore a wizard which generates the model can be used or it can be written manually. Either way the model has to match the requirements defined in section 6.2.2 Create a model for a configuration. Based on the Ecore metamodel it is now possible to create the EMF genmodel which generates all the needed classes for GMF.

By following the descriptions of section 6.2.3 Build a graphical editor for a model the gmfgraph, gmftool and gmfmap files needed by GMF are created. Each file defines a part of the editor. Now generate the editor and use it for writing development models.

Figure 7.9 shows the different steps from the XSD to the editor.



```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:element name="Class">
    <xs:complexType>
      <xs:choice>
        <xs:element ref="IncludesClass" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="ImplementsClass" minOccurs="0"/>
      </xs:choice>
      <xs:attribute name="name" type="xs:string" use="required"/>
      <xs:attribute name="version" type="xs:string" use="required"/>
    </xs:comp    ype>
```

Figure 7.9: From XSD to an Eclipse Modeling Framework model to a Graphical Modeling Framework based editor

The structure of the development model created using the editor is not compatible to the one defined by the XSD. That's why the file created by the editor has to be mapped to an XML file based on the XSD. JDOM will be used for this purpose.

JDOM for mapping the model to XML

As previously described a JDOM transformation of the development model will be done. The section 3.4.2 JDOM - Java Document Object Model explains the details of this approach. There are some development skills necessary to write the mapping, because the structure of the file is very different to the one described by the XSD.

After transformation, the file describing the development model will be valid for the XSD listed above. The example configuration shown by figure 7.8 will be described by the XML file listed below.

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="flightplan" path="trunk/examples/flightplan" version="1.0">
  <package name="persistence" version="1.0">
    <class name="AerodromStore" fullname="peristence.AerodromStore" version="1.0">
      <includesClass name="AerodromImpl" package="common" />
      <includesClass name="PositionImpl" package="common" />
    </class>
    <class name="BeaconStore" fullname="persistence.BeaconStore" version="1.0">
      <includesClass name="AerodromStore" package="persistence" />
      <includesClass name="AerodromImpl" package="common" />
      <includesClass name="PositionImpl" package="common" />
      <includesClass name="BeaconImpl" package="common" />
    </class>
    <class name="DBController" fullname="peristence.DBController" version="1.0">
      <includesClass name="DBManager" package="persistence" />
      <includesClass name="BeaconImpl" package="common" />
      <includesClass name="FlightImpl" package="common" />
      <includesClass name="PositionImpl" package="common" />
      <includesClass name="TrackpointImpl" package="common" />
    </class>
    <class name="DBManager" fullname="peristence.DBManager" version="1.0" />
  </package>
  <package name="common" version="1.0">
    <class name="Trackpoint" fullname="common.Trackpoint" version="1.0" />
    <class name="TrackpointImpl" fullname="common.TrackpointImpl" version="1.0">
      <implementsClass name="Trackpoint" package="common" />
    </class>
    <class name="Aerodrom" fullname="common.Aerodrom" version="1.0" />
    <class name="AerodromImpl" fullname="common.AerodromImpl" version="1.0">
      <implementsClass name="Aerodrom" package="common" />
    </class>
    <class name="Position" fullname="common.Position" version="1.0" />
    <class name="PositionImpl" fullname="common.PositionImpl" version="1.0">
      <implementsClass name="Position" package="common" />
    </class>
    <class name="Beacon" fullname="common.Beacon" version="1.0" />
    <class name="BeaconImpl" fullname="common.BeaconImpl" version="1.0">
      <implementsClass name="Beacon" package="common" />
    </class>
    <class name="Flight" fullname="common.Flight" version="1.0" />
    <class name="FlightImpl" fullname="common.FlightImpl" version="1.0">
```

```
      <implementsClass name="Flight" package="common" />
    </class>
  </package>
</project>
```

Since Model Driven Software Development (MDSD) functionality will be added to our prototype, the next section is about generating the Java classes and packages using openArchitectureWare (oAW).

openArchitectureWare to generate the application skeletal

By integrating MDSD, the benefit of MDCM and of modeling software systems is raised. The development model describes the artefacts used for development and provides the foundation for MDSD. The more detailed the model is, the more can be generated. OpenArchitectureWare will be used as MDSD framework. Section 3.4.1 openArchitectureWare explain oAW in greater detail.

In this prototype all classes including their import/implement statements and the package structure will be created. This functionality can be easily achieved with oAWs workflow and template technology. Of course, as the amount of generated artefacts rises, so does the complexity of the transformation.

The next chapter shows how to use the prototype and MDCM in order to ease development.

Using the prototype

As a result of the previous sections, a running and working editor for MDCM development models is available. This prototype only provides the basic functionality of MDCM. The features supported and not supported by the prototype are listed below.

Prototype features

+ Package view of the development model of MDCM
+ Generate an XML input for CM
+ Generate Java classes and the application foundation
– Support of the other development model views of MDCM
– Support of the software, requirements and release models of MDCM
– More MDSD functions like C# code generation, data base integration and more

A full implementation of MDCM can do much more for the development process. By supporting different programming languages and additional database integration, MDCM can be used for every project.

Even though the prototype only provides a small part of MDCMs features, it can be used to compare the traditional approach with MDCM. The traditional approach requires manual writing of the Java classes and manual definition of the CM items. When using the prototype only the development model has to written, which will not only be used to generate the application skeletal, but also for communication purpose. In many projects, a kind of development model will be created using UML, but without using it for generating further components. So MDCM delivers the same benefit as MDSD plus the additional benefit of the CM integration.

7.2.5 Model Driven Configuration Editor support for the flightplan configuration files

The MDCE will beused to support writing and modifying of the most important configuration files of the flightplan example. The first file, an editor is written for, is the eventserver.xml, which describes the configuration of the Naiad event server used by the flightplan example. MDCM provides an editor to write a model of the configuration and transforming it to an XML file. The second file is the eventserver-mule-config.xml. It describes the configuration of Mule including the whole workflow of the application. Two different editors are provided to describe the Mule configuration, one showing the XML view of the configuration and one displaying the workflow. Both XML files can be found in appendix B9 and B10.

Necessary steps to create an Model Driven Configuration Editor

The section 6.2.1 Model Driven Configuration Editor - How to create an editor for configurations describes the necessary tasks to create MDCEs. The creation of all three editors follows this approach. There are XML Schemas for both configurations which provide the foundation for building the meta-model of the editor. Then the EMF components are created, the GMF editor is written and modified and the transformation of the model to the XML file is developed. The whole approach including the used technologies is described in 3 Related work and applied technologies. When using the MDCE approach, the creation process of the editors shouldn't take too long.

The Naiad event server/MuleConfig editor

Both editors have been built according to the MDCE approach. Both editors are Eclipse Plug-Ins based on EMF and GMF. JDOM is used for the model-to-configuration transformation. The editors provide the functions of GMF and offer a quick and easy way to write models and to generate configuration files. The next page shows two screenshots of the editors. As can be seen, each XML tag is represented by an object and the nesting of the tags is modeled by connections. Both editors support the users in creating the configuration file. After the screenshots, the workflow view of the Mule editor is described.

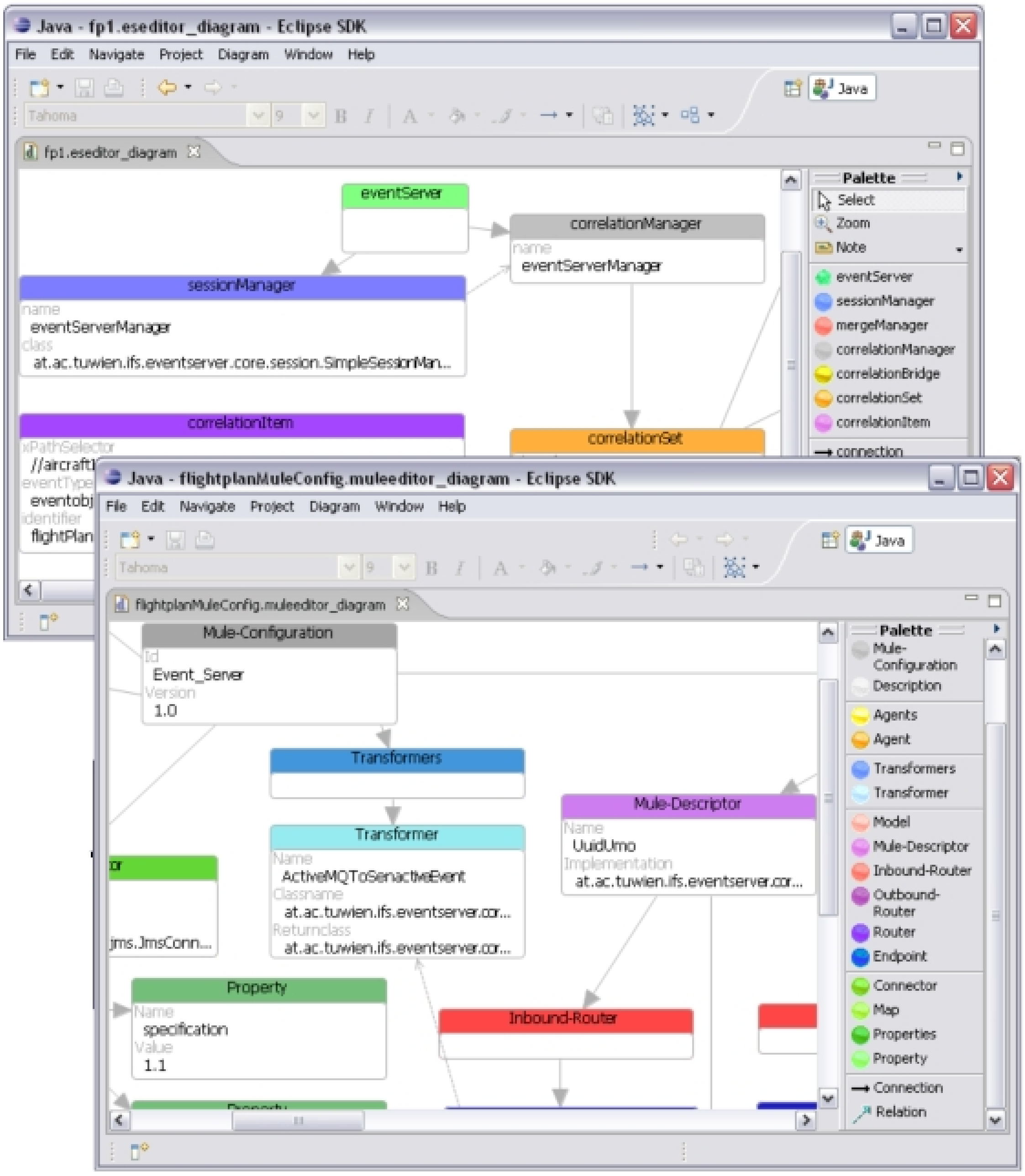


Figure 7.10: Model Driven Configuration Editor for Naiad event server configuration

The MuleConfig workflow editor

The Mule workflow editor differs from the previously described editors. He doesn't show the structure of the XML file and he also cannot be developed using an XML Schema as foundation. The Mule workflow editor models the application flow of a Mule application. The editor shows the connections between the UMOs and the running sequence of them. This example illustrates another advantage of MDCE over common XML editing: The view of a model of the editor can be chosen and modified to actually describe the application in order to meet the own needs and at the same time, the model can be mapped to the XML configuration.

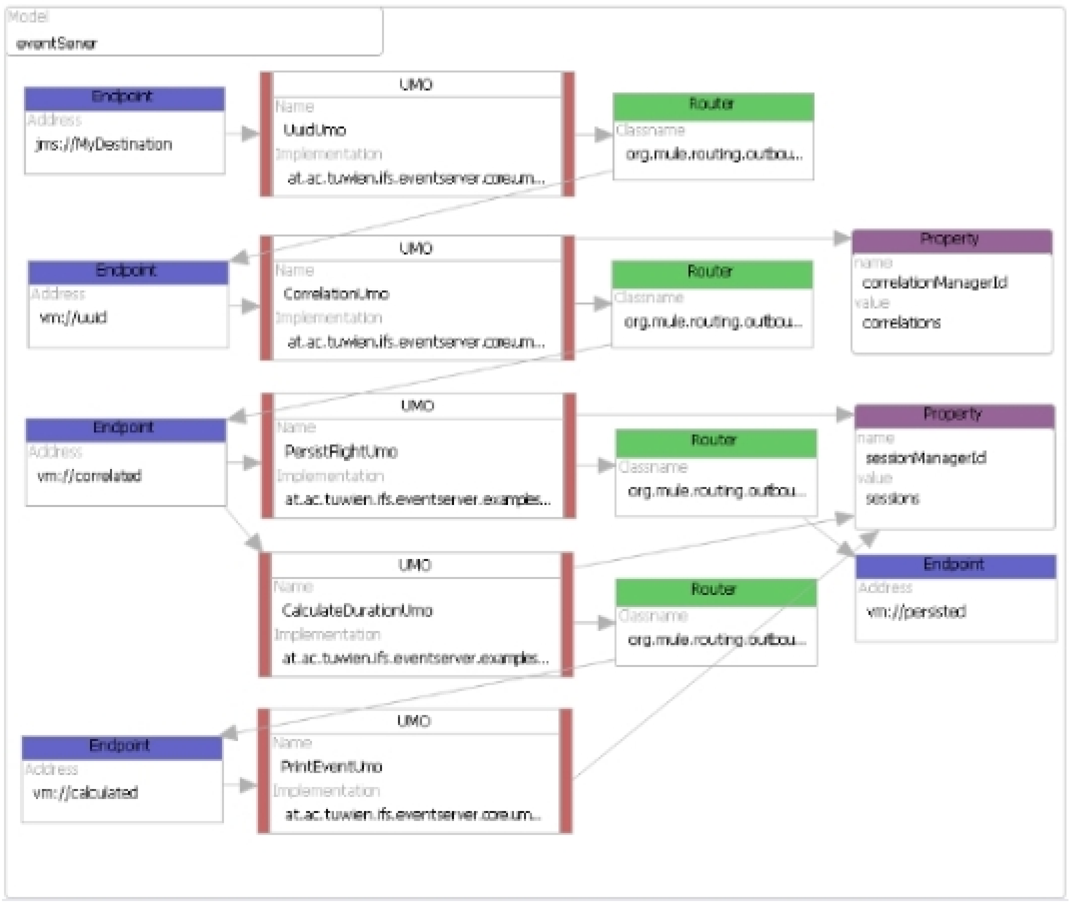


Figure 7.11: Core of the Mule workflow model of the flightplan example

The procedure model for creating such a modified type of an MDCE is the same as usual. Partly defining the metamodel for the editor is more complex. A metamodel which fits the custom view has to be modeled. The second task that grows in complexity compared to the standard MDCE creation process is the development of the mapping procedures. The more the custom view differs from the XML representation, the more complex the mapping will be.

Figure 7.11 illustrates a Mule workflow model written using the MDCE. It defines the procedure of the flightplan example and generates the exact same configuration as the Mule editor does.

This chapter has shown examples and screenshots of MDCEs for a particular application. The chapters 8 Results and 9 Discussion will deal with the open issues and results of this work.

8 Results

While working on Model Driven Configuration Management (MDCM), much knowledge has been gained and some new approaches have been invented. But the main goal to achieve a merge of configuration management and software development turns out to be too complex and difficult. This chapter compares the achievements of MDCM with the targets of it. The failed targets and open issues are described and alternative approaches are analysed. Additionally future enhancements and expansions are introduced.

8.1 Goals of Model Driven Configuration Management

Besides the implementation of the target application, most software development (SD) processes focus on creating a large and comprehensive documentation. Since documentation has to be done anyway, these documents can also be used to describe the target system in detail so that parts of the application can be developed only with the informations of them. Model Driven Software Development (MDSD) is based on this fundamental idea and provides techniques like metamodels and specified documentation artefacts in order to generate code from the models of the documentation. But the MDSD approach doesn't not cover development support processes like configuration management (CM). And that's where MDCM enhances the MDSD.

The main target of MDCM can be described in one sentence: MDCM tries to significantly reduce redundancy through combining software development and configuration management.

MDCM tries to use the similarities of CM and SD to develop general models to describe the artefacts and parts of CM as well as of SD. Based on this idea, the main goals and requirements have been decided.

MDCM should offer a procedure model that supports the whole SD team on developing software with integrated CM based on the MDSD approach. On the other side, a flexible and continuous tool support for MDCM should be developed. In order to ease the whole process of creating MDCM, it has been separated into the procedure model and the tool support. Also the goals have been separated.

On the one hand goals for the procedure model of MDCM have been specified and on the other hand the goals for the subproject Model Driven Configuration Editor (MDCE) have been defined. Even separated, the main goal and reason for both projects was still to reduce redundancy. The targets of each part are described below.

Goals of MDCM

- Merge CM and SD where it's possible and benefit from the similarities.
- Create a well defined procedure model for the whole SD process which integrates CM as a major part of it.

- Enhance the collaboration and communication between the roles of CM and SD.
- Provide standardisation through metamodels in order to allow automatic tooling.

Goals of MDCE

- Support MDCM.
- MDCE should be a continuous, extendable tool for modeling and transforming configurations.
- Provide input for the standard CM tools.
- Generate code and serve as an MDSD tool.

Another target which appeared during the first researches was to enable the MDCE to model XML based configurations. More general, to support XML as a language to define models and metamodels. So the MDCE should also become a graphical tool for writing and modifying configuration files.

When reading this list of big and complex goals, it's getting obvious that not all of the goals could be reached during the work on MDCM. But still plenty of new very useful approaches have been discovered and developed by MDCM. The next section describes the achievements of MDCM and also the issues which couldn't been solved.

8.2 Results of Model Driven Configuration Management

This section describes the results, achievements and open issues of MDCM. Like the last section, this one is also separated into two parts, MDCM and MDCE. The goal of this chapter is to summarize the features of MDCM and MDCE and to show, which of the requirements have been realised and which are still open issues. This section doesn't evaluate MDCM on its relevance in praxis. This is done by the next section.

The Model Driven Configuration Management approach as a procedure model

There are many possibilities to merge configuration management (CM) and software development (SD). MDCM tries to support the development team with a best practice approach. In order to be flexible enough for almost any kind of appliance, MDCM is a very general approach. It defines the major steps of SD, describes the roles and an integration of CM through metamodels. Sadly this general approach has major drawbacks. MDCM cannot be used as a procedure model. Instead it has to be combined with one. It also cannot be fully applied as it is. Because of the huge requirements on such a general approach, MDCM covers many topics but none of them in detail. So to really use MDCM, it has to be further specified. But on the other side MDCM isn't limited on a specific domain and it provides many ideas and approaches for future research.

The metamodel approach of Model Driven Configuration Management

As a foundation for the integration of CM and SD, MDCM introduced the configuration management metamodel language (CMML). CMML is a collection of metamodels which define the artefacts of a software development process and also their relations. Each stage of the development process has at least one metamodel. The models based on the CMML metamodels define the concrete artefacts and the connections to and dependencies on other artefacts. These artefacts are the configuration items (CI) for the software configuration management. An obvious advantage of creating these models is that they provide a great overview of the artefacts which are either already existent or have to be created in future. The CMML specifies the artefacts and their allowed relations. Because of the exact definition offered by the CMML, it is now possible to create a tooling to write and modifying these models and to generate XML definitions of the artefacts which can be used as input for even further processing.

The metamodels for the analysis, design, test and deployment stage only define the documentation artefacts. The contents of the documentation has to be described without the support of MDCM. Because of that it's not possible to use the content as an input for further automatisation. Of course it would be more efficient to include the contents of the artefacts to the MDCM process. But it was too time-consuming and complex to integrate all possible UML models and other diagrams to MDCM. Also it was regrettably not feasible to describe each stage in detail.

Instead MDCM and the CMML focused on the most important stage of the SD process, the implementation stage. The implementation stage uses the artefacts of the previous stages to develop the application. MDCM supports the implementation stage by special, class based CMML diagrams. The CMML application model provides configuration management specific informations as well as development specific informations. This model is the heart of MDCM. It provides all necessary informations to generate the basic application code skeletal and to generate an XML representation of the configuration items of the application. Even though that generation works fine, there is a major drawback. The crucial issue of how to import the XML representation into CM tooling couldn't be solved. Thus it is nice to generate the XML file but only occasionally useful. But nonetheless the CMML application model with its package and class view provides an excellent overview and graphical representation of the components and relations of the application. And of course, the MDSD code generation is also possible with the CMML application model.

MDCM provides different new approaches for each stage of the development process. But it doesn't offer a continuous end to end support. The different models are connected, but still many parts of the process are not related. Although MDSD is a part of MDCM and is used in the implementation stage, in order to be able to use MDSD throughout the whole process, the MDCM approach has to be further extended towards common MDSD approaches.

The Model Driven Configuration Editor as a tool support

When thinking about applying the MDCM approach, it's obvious that tool support is very important for the success of MDCM based software development processes. That's why the MDCE has been created. At the beginning the main purpose of it was to be a tooling for MDCM. But as can be seen later, MDCE has evolved from an MDCM tooling to a universal modeling tool for XML based configurations.

MDCE supports MDCM by providing an efficient way to generate an editor for the models of the CMML. An MDCE is based on a concrete CMML metamodel and applies its rules and structure. This means the models designed with an MDCE are always valid models. All elements and connections available in the editor are defined by the metamodel. Further it is possible to develop transformers from model to code or XML. MDCE can provide all this functionality, but it's not a ready to use software package. There are some prototype implementations, but the MDCE is more a manual how to create an editor. This approach makes it possible to develop an MDCE for nearly every purpose. On the other hand there is no continuous tool support. And one major drawback of the MDCE is that it's a collection of many small editors and it's very hard to integrate them in one large tool. So rated on its qualification as a tooling support for MDCM, MDCE is not the best solution. But the next section shows the real potential behind the MDCE.

The Model Driven Configuration Editor for configurations

When developing the MDCE, it was one goal to create an editor which is really flexible and easy to enhance. The first attempts always resulted in editors providing some functionality very good, but other not at all. So the considerations have gone away from creating an application towards providing a manual to create an editor for the particular requirements.

An editor developed after this manual should be able to create models for all different kinds of configurations, with only one condition, the output has to be an XML file. Since XML Schemas define the structure of XML files, the manual contains a description how to create a metamodel out of an XSD. With this and some other XML specific enhancements the MDCE starts to became also a manual describing how to build a graphical editor for XML files.

When using MDCE to create an editor for an XML file, the metamodel can be generated using an XML schema. The next step is to define the graphical representation. Additionally the mapping from the MDCE model back to the XML file has to be done. This is the most effort because a transformer has to be developed. But after that a comfortable editor for an XML file based on its specific XSD is available. One prototype editor developed during the MDCM process has been built do create and modify configuration files of the mule framework. There, the MDCE is not only used to provide a graphical editor for the XML file, it is also able to represent the structure of a Mule process itself. So with this editor it is possible to design the mule process structure and the according XML file for this process is generated. This appliance shows that very complex editors can be created with MDCE with relatively small effort.

On the other hand there are still some open issues. The MDCE doesn't support the generation of a model on foundation of an existent XML file. It only allows to design a model and create the XML file. It's not easy to add this feature, but it should be one of the first enhancements of the MDCE. Also additional validation is not planned. Another issue is, that changes in the metamodel for the MDCE require a complex adaptation of the MDCE to support them.

Features and limitations of the Model Driven Configuration Editor

In order to enable an easy and quick creation of an editor providing all the required features, MDCE is supported by some powerful frameworks. MDCE uses the Eclipse Modeling Framework (EMF) for defining the models and the Graphical Modeling Framework (GMF) for the graphical representation. Because of using those frameworks, the MDCE is not only able to use their features, it also shares the same limitations. Due to the fact of the specialisation on EMF, UML is still supported but EMF is recommended. When using UML as the modeling language, development is more difficult because the GMF needs an EMF model as input. So the UML model has to be transformed to an EMF model. Even though the transformation can be done automatically, it's still an additional step which may cause troubles.

There are some limitations when modeling an XML file with EMF or UML. Not every condition offered by an XML Schema (which describes the XML file) are also available for a metamodel (which describes the model). These conditions have to be implemented manually and without framework support.

The graphical representation of the MDCE is done by the GMF. The GMF offers a lot of functionality out of the box and it is developed for exactly the purpose MDCE uses it. It provides actions like resizing, sorting, drag and drop and other features which can be used to display and handle elements and connections. Usually the features of the GMF provide all the necessary functionality. But if it doesn't offer a feature from the start, it's very hard to add it. Enhancing the GMF requires major Java development skills.

OpenArchitectureWare (oAW) is used to develop the transformers. oAW is a MDSD framework which allows to easily create transformations of models. Sadly it is very hard to integrate oAW to the editor. Because of that, JDOM is also supported as an alternative to implement transformations. JDOM is a Java library to process XML documents. There are no limits in using JDOM and it also can be integrated to the MDCE easily. On the other side the development of the transformer is relatively time-consuming and complex and JDOM does not process the models. Instead it works with the XML representation of them.

8.3 The future of Model Driven Configuration Management

Even though there have been successful projects using the MDSD approach, most projects still don't use MDSD. Many experts think that the MDSD approach is to general to be efficient. A new trend in software development presented on the OOP 2008 [1] is to apply the MDSD approach for an explicit small domain. Because of the small application area, the MDSD activities can be specialised for this area and are thus more efficient. MDCM applied for such a small domain might be really powerful. MDCM could enhance the MDSD activities and thus provide even more functionality. On the other side, configuration management can also be seen as a kind of domain and MDCM could be further enhanced towards CM features. The most important part of these advancements would be to enable a close collaboration between MDCM and the CM tooling, especially to make it possible to import MDCM models to CM tooling. MDCM provides many approaches and some prototype implementations and it delivers a lot of potential, but it has to be further enhanced. Another possible way to do this is to provide a complete set of CMML metamodels and models with continuous tool support based on the MDCE.

The MDCE is already in a state where it can be used in real world projects, although it has still potential

[1] http://www.sigs-datacom.de/sd/kongresse/oop_2008/index.php

for enhancements. One of them would be the automatic generation of an MDCE model for an existent XML, which could be called as reverse engineering, since the generation ot the XML file out of its model representation is the standard case. Also more validation possibilities would be really useful for the MDCE. On the other side its possible to imagine a complete automatic creation of the MDCE based on an XSD. This approach might even lead to a tool which generates MDCEs by getting an XSD as input.

In summary it has to be said, that although many parts of MDCM are still in a prototype state, the approach of MDCM has a lot of potential for the future.

The next section compares the software development with and without MDCM and tries to determine the value of MDCM for real world projects.

9 Discussion

This chapter compares the Model Driven Configuration Management (MDCM) with the traditional software development (SD). The differences between the approaches are listed and the assets and drawbacks of each procedure are described. This chapter also explains the characteristics a project should have so that it's recommendable to use MDCM and which projects should rather use the traditional SD.

9.1 Software Development versus Model Driven Configuration Management

This chapter summarises the major attributes of the related approaches and also describes the pros and cons of each. Additionally to SD and MDCM, Model Driven Software Development (MDSD) is also compared with the others. Because of the potential of MDCM and the fact that the approach is still in further development, the MDCM which is described in this work and the MDCM which could be developed in future are compared to SD and MDSD.

9.1.1 Traditional Software Development

The traditional SD process is still the approach mostly used for creating a software solution. There are many modifications of it, but the general activities are more often than not the same. The different approaches reach from extremely strict defined processes like the V-Model XT to the agile development methods. For this chapter in order differ them from the MDSD approach, all of them are referred to as traditional SD processes. What all these approaches have in common is that they do not define the use of configuration management (CM). They also do not generate code or models from other models. The documentation and the implementation are separated, which leads to the following pros and cons of the traditional SD approaches compared to the others.

Qualities of SD compared to the others:

+ Great flexibility through different approaches like Extreme Programming or Agile SD.
+ No compulsory artefacts, concentration on the essential artefacts.
+ No dependency on artefacts, the application is the center of attention.
+ Faster iterations are possible.
+ Smallest effort on creating and modeling and documentation.
– Larger effort in maintaining the documentation than in MDCM.
– Redundancy between the artefacts is very large.

– Inconsistency nearly unavoidable.
– Collaboration between CM and SD not ideal.

9.1.2 Model Driven Software Development

Because of the great relevance of MDSD for MDCM, it is compared separately and not as a part of the traditional SD processes. MDSD has the focus on modeling the important parts of the target application and generating code out of the models. MDSD works a lot with transformers and different stages of models. The MDSD inherits most attributes of the traditional SD process, but through its specialisation on models the following aspects are different.

Qualities of MDSD compared to the others:

+ Well defined procedure.
+ The major models and general steps are described.
+ Less effort in implementation through generation.
– More effort in modeling the system.
– Not as flexible as the traditional SD processes.
– Including own code into the generated is always problematic.
– Modeling application logic is not always possible and often hard.

9.1.3 Model Driven Configuration Management - Now

MDCM tries to enhance MDSD by integrating CM to the model driven approach. Even though MDCM is a very open approach which can be combined with many traditional SD processes, it is not capable of being used with agile SD approaches. That is because of the focus of it on the artefacts. There is some overhead when using MDCM which should be compensated by less redundancy and inconsistency. But since the MDCM approach has still some open issues which affect its usage, not all goals of MDCM could be achieved up to now. But still there are some really usable and relevant features which makes MDCM a good choice for some projects. The major pros and cons of the available MDCM approach are listed next.

Qualities of MDCM compared to the others:

+ The important artefacts are defined by CMML.
+ Provides a good overview of artefacts and configuration items.
+ MDCM delivers a general best practice approach for handling artefacts.
+ Generates implementation skeletal out of the CMML models.
+ Generates an XML representation of the configuration items used as input for CM toolings.
+ Collaboration between CM and SD profits from the shared models.
– Import of the XML representation for CM toolings is not available out of the box.
– More effort in modeling the system.

- – Less flexible than traditional SD.
- – Has to be combined with a fitting procedure model.
- – Many parts are in a prototype state.

9.1.4 Model Driven Configuration Management - In the future

As mentioned before, there are some possibilities to further advance MDCM. The enhancements defined in this section are all not particularly complex, but most of them require some further work and also research. In the future MDCM, the CMML is enhanced to a complete list of all necessary artefacts including their relations. MDCE is extended to provide continuous tool support all offered by one Eclipse Plug-In. Selected CM tools have been enhanced to be able to import the XML representation of the configuration items and define them and their relations according to the XML file. With these enhancements, even more projects should be able to benefit from MDCM. The characteristics of the future MDCM are listed below.

Qualities of a MDCM in the future compared to the MDCM now:

- + All artefacts are defined by CMML.
- + Collaboration between CM and SD is optimized.
- + Less redundancy and inconsistency.
- + Import of XML representation to CM toolings.
- – Still more effort in modeling the system.
- – Still less flexible than traditional SD.
- – Still no complete procedure model.

9.2 The right approach for each project

MDCM is not suitable for every project. Like MDSD, MDCM requires large or at least not small projects to unfold its qualities. Often small project have no or only rudimental configuration management, which disqualifies MDCM for those projects. Also the software developed by the project is relevant for the selection of the approach. MDCM and MDSD are not qualified for projects with a focus on process flows, sequences and logical workflows. Both are better suited for projects with large databases, powerful classes and other large static components. That's because these components are way easier to model than the action oriented.

Also it is problematic if the requirements are changed often or the project requires dynamic structures. Because of the focus of MDCM and MDSD on artefacts and its relations, changes in one artefact require also changes in the related artefacts. An advantage of MDCM is, that the CMML defines those relations and thus they can be easily retraced. On the other hand, there are many static artefacts which are rather counterproductive if the project requires dynamic actions and structures.

Another criteria is the competence and knowledge of the project staff. Unexperienced developers should use the traditional SD process, because its the best way to get into professional development. MDSD and

MDCM can be notably useful if the analysts and designers of the project are gifted in modeling. Table 9.1 gives an overview on the different attributes of a project and how suitable the approaches are for each.

Criteria	**SD**	**MDSD**	**MDCM**
Large project	++	++	++
Small project	++	o	-
Action focused	++	-	o
Structure focused	+	++	++
Static project	+	++	++
Dynamic project	++	-	o
Unexperienced Staff	+	-	-
Modeling specialists	o	++	+

Table 9.1: Project characteristics and the fitting approach
Conventional Software Development (SD)
Model Driven SD (MDSD)
Model Driven Configuration Management (MDCM)
- unqualified, o partly qualified, + well qualified, ++ great qualified

The traditional SD is suitable for most projects. Often the right SD procedure model has to be selected, but there is a wide range of different SD approaches to choose from. Some projects can really benefit from the MDSD approach and also from MDCM. For example it could be really efficient to choose MDCM for a large static project with a focus on static components.

10 Summary and further work

10.1 Conclusion

This chapter provides a summary of all covered topics of this work. Even there was not enough time to solve all problems and to test Model Driven Configuration Management (MDCM) in a practical environment, a lot of research had been done. The first section provides information on the results and open issues regarding MDCM, the second section offers the same information about the Model Driven Configuration Editor (MDCE) and the third section gives a summary of the used technologies.

10.1.1 Configuration management and modeling

This work introduces the concept of MDCM. Since there is only one known small article about this topic, most of the MDCM part is new. When starting to work on this part, it was obvious to use the Unified Modeling Language (UML) as modeling language and to try to connect MDCM with Model Driven Software Development (MDSD). The main issue was to find a configuration management (CM) tool which supports the import of configurations, either as an XML file or better as a UML model. This is unfortunately still an unsolved issue. It was also very hard to keep the models clear because even small applications have a huge amount of configuration items.

Results of MDCM

- Modeling of configurations is possible and reasonable, but the insufficient tool support makes a practical use of MDCM nearly impossible.
- The many different definitions of CM make it hard to extend CM. Throughout the writing process of this work, a definition of the parts and the management disciplines of CM has been created. Based on those definitions, prototypes of metamodels describing the different sections of CM have been created. Metamodels for the analysis, design, development, test and deployment stages are available.
- MDCM models can provide a detailed definition of CIs and can therefore be used as input for change management.
- A merge of CM and MDSD through MDCM models can be realised. Already tools like the Rational ClearCase provide MDSD and CM in one tool. Both approaches need different views of the system which can be achieved through transformation. Main problem is still the import of a model into the CM part.
- Unfortunately models become very large when using them to describe configurations. Therefore, different views have been developed.
- A procedure model which shows how to execute MDCM the right way has been written.

Open issues of MDCM

- Since none of the studied tools support the import of configurations in form of an XML file, testing the MDCM workflow including tool support was not possible.
- Develop an enhancement for an established open source CM tool to provide import of configuration models.
- Contact major vendors in order to get their opinion on MDCM and in order to get them thinking about a possible tool support.
- Develop a whole project using the MDCM approach, including the creation of models for the different stages.
- Write models which can be used as a foundation for CM and MDSD. Map the models to the specifics views of MDSD and CM.
- Extend the metamodel prototypes.

10.1.2 Editing of configurations

Model driven configuration editing (MDCE) is an approach to create efficient editors for configurations. One main requirement was to develop a process model for creating an editor for XML configuration files. The whole creation process should include as less coding as possible but still provide the flexibility to write editors for nearly any kind of configuration. This target was achieved by using Eclipse and the Plug-Ins Eclipse Modeling Framework (EMF) for modeling and Graphical Modeling Framework (GMF) for the user interface. An important feature of the editor is the mapping of the written graphical model back to the original configuration file. At first, openArchitectureWare (oAW) had been used to transform the model. The transformation worked fine, but running the oAW workflow as Plug-In didn't work. There is a way to run the workflow as a Plug-In by starting an additional Java virtual machine, but there were unsolvable dependency problems. In order to offer the mapping as a Plug-In, JDOM mapping was included to the MDCE project. Since the model and most configuration files are saved as XML files, JDOM transformation works fine.

Results of MDCE

- A procedure model has been written in order to ease the development of GMF based configuration editors.
- The combination of Eclipse and the Plug-Ins EMF and GMF enables to easily create graphical editors.
- The creation of the EMF model and the representation by GMF have been standardized in many ways. XML Schemas can be mapped into an Ecore model using already existent rules and the look of GMF when presenting a model based on an XML file has been defined in this work.
- The editor can be used as an Eclipse Plug-In. Unfortunately, EMF and GMF have to be installed on the Eclipse environment, which luckily is a lot easier since the Callisto framework has been released.
- The mapping of the created model to an XML file has been reached by using oAW or JDOM. OAW has the advantage to do a real model-to-code transformation, but its workflow cannot be started as a Plug-In. JDOM instead simply transforms an XML file to another and is additionally runnable as a Plug-In.

Open issues of MDCE

- The models created by MDCE can be transformed to XML files. As an additional feature, converting an XML file to a model would be very useful. The transformation of the XML file to the data file of the model is no big problem. But each GMF model has also a file which saves the graphical data. To generate this file is possible, but very time-consuming and complex.
- When modeling the flightplan example, two editors (each providing a different view) for one configuration have been created. A mapping between those two views or even better, an editor with one data file and two different graphical presentations would really ease modeling.
- The oAW workflow cannot be started as a Plug-In.
- The procedure model and the transformation rules have standardized the creation of an editor based on an XML Schema. The next step would be to automate the creation and generate an editor. Even it is not that easy to implement, it's feasible to do.

10.1.3 Used technologies

When thinking about how to create an editor, the first decisions considered the used technologies. After trying the Graphical Eclipse Framework (GEF) which points out to be too complex, GMF was selected as the graphical framework. Since GMF needs EMF to provide the model, the modeling language for MDCE has to be EMF. Both frameworks, EMF and GMF, offer editors and generators as Eclipse Plug-Ins. OpenArchitectureWare provides the model-to-configuration transformation and JDOM XML-to-XML mapping.

Results concerning the used technologies

- When evaluating both graphical frameworks, GEF tends to be more complex but also provides more functionality. GMF is specialized in creating models and is therefore better suited for MDCE.
- OpenArchitectureWare is growing to a big and important MDSD tool. The whole transformation process and the template language are well designed. OAW was used for model-to-model and model-to-code transformation.
- The DOM API of Java 5 and JDOM have been evaluated and JDOM seems to be the more consistent than the standard DOM API.

Open issues concerning the used technologies

- As mentioned in the previous section, the problem with running an oAW workflow as Java Plug-In hasn't been solved.
- Some of the GMF layout types didn't work at all or tend to behave irreproducible. Also some parts of GMF could have a better documentation.
- Additional features can be added to GMF. Extending the GMF methods is necessary to do this. Unfortunately, there wasn't enough time to enhance GMF during this work.

10.2 Outlook

This section tries to offer a foresight on management disciplines and technologies related to this work. Many frameworks and methods used in this work are at the beginning of their life cycle and are still further improved. The whole software development business including tools and procedure models changes fast. It's very hard to predict upcoming changes, still some trends are so big that they will maintain for a time. This section not only tries to predict the future importance of MDCM and MDCE, it also looks at the used frameworks and software development methods.

Model Driven Architecture / Model Driven Software Development

Model driven architecture (MDA) and Model Driven Software Development (MDSD) are very interesting development approaches. Even only a small percentage of applications are developed based on MDA/MDSD, the share still gets bigger. The main definition of MDA/MDSD is nearly complete, the most research and improvement is done at the tooling sector. The first efficient and technically matured open source MDSD frameworks are already available and support is getting better. As discussed in this work, a future research area might be the merge of MDA/MDCM and CM tools. Also a new research area is to use the MDSD approach for small domains. The first projects implemented with this approach where a huge success. An detailed description of common state-of-the-art SCM tools can be found at [LEON 2005].

Configuration Management

Configuration management (CM) is a very important management discipline for every big project. Most developers and managers cannot even imagine to work without the support of CM tools anymore. In future the features of CM tools will be enhanced and many of the different small tools (requirement/release tools) will merge to big suites providing all functionality. Unfortunately it is rather unlikely that CM tools will be enhanced to offer the import of models for defining configuration items.

Model Driven Configuration Management

Modeling configuration items can be very helpful. MDCM combines modeling of CIs with MDSD and provides metamodels as a foundation. In order to be able to use MDCM as a configuration management approach, some further steps have to be done:

- Enhance the metamodel prototypes to get a solid foundation for tool support.
- Support for MDCM has to be provided by a CM tool.

- Refinement of the procedure model.

MDCM can achieve very efficient modeling and development especially in combination with MDSD and CM tool support. But further improvement on MDCM and on the tool support is a precondition for using MDCM professionally.

MDCE

MDCE already provides a solid procedure model for creating configuration editors. But some improvements would make MDCE even better:

- Providing a roundtrip generator to generate a model of an XML file.
- Different views for one data file.
- Automated editor creation.

MDCE is really a comfortable and quick way to create editors. By further improvement of GMF and of the procedure model, MDCE can get even more powerful.

A Code samples

A.1 EventServer.ecore

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="EventServer"
    nsURI="http://EventServer" nsPrefix="EventServer">
  <eClassifiers xsi:type="ecore:EClass" name="Diagram">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"
      eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="author"
      eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="eventserver"
      lowerBound="1" eType="#//ES" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="sessionmanager"
      lowerBound="1" upperBound="-1" eType="#//SessionManager" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="mergemanager"
      lowerBound="1" upperBound="-1" eType="#//MergeManager" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="correlationmanager"
      lowerBound="1" upperBound="-1" eType="#//CorrelationManager"
      defaultValueLiteral="" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="tuplecorrelation"
      upperBound="-1" eType="#//TupleCorrelation" containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="ES">
    <eStructuralFeatures xsi:type="ecore:EReference"
      name="eventserversessionmanagerrelations" upperBound="-1"
      eType="#//EventServerSessionManagerRelation" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference"
      name="eventservercorrelationmanagerrelations" upperBound="-1"
      eType="#//EventServerCorrelationManagerRelation" containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="SessionManager">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="sessionname"
      eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference"
      name="sessionmanagermergemanagerrelations" lowerBound="1"
      eType="#//SessionManagerMergeManagerRelation" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference"
      name="sessionmanagercorrelationmanagerconnection" lowerBound="1"
      eType="#//SessionManagerCorrelationManagerConnection" containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="MergeManager">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="mergename"
      eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="CorrelationManager">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"
      eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference"
      name="correlationmanagertuplecorrelationrelations" upperBound="-1"
      eType="#//CorrelationManagerTupleCorrelationRelation" containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="TupleCorrelation">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="identifier"
      eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="EventServerSessionManagerRelation">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"
      eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference"
      name="source" eType="#//ES"/>
    <eStructuralFeatures xsi:type="ecore:EReference"
      name="target" eType="#//SessionManager"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass"
    name="SessionManagerMergeManagerRelation">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"
      eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference"
      name="source" eType="#//SessionManager"/>
    <eStructuralFeatures xsi:type="ecore:EReference"
      name="target" eType="#//MergeManager"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass"
    name="SessionManagerCorrelationManagerConnection">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"
      eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference"
      name="source" eType="#//SessionManager"/>
    <eStructuralFeatures xsi:type="ecore:EReference"
      name="target" eType="#//CorrelationManager"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass"
    name="CorrelationManagerTupleCorrelationRelation">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"
      eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference"
      name="source" eType="#//CorrelationManager"/>
    <eStructuralFeatures xsi:type="ecore:EReference"
      name="target" eType="#//TupleCorrelation"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass"
    name="EventServerCorrelationManagerRelation">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"
      eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference"
      name="source" eType="#//ES"/>
    <eStructuralFeatures xsi:type="ecore:EReference"
      name="target" eType="#//CorrelationManager"/>
  </eClassifiers>
</ecore:EPackage>
```

A.2 EventServer.gmfgraph

```
<?xml version="1.0" encoding="UTF-8"?>
<gmfgraph:Canvas xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
```

```
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:gmfgraph="http://www.eclipse.org/gmf/2005/GraphicalDefinition"
  name="EventServer">
<figures name="Default">
  <figures xsi:type="gmfgraph:RoundedRectangle"
      referencingElements="ES" name="ESFigure">
    <children xsi:type="gmfgraph:Label"
        name="ESLabelFigure" text="Event Server">
      <foregroundColor xsi:type="gmfgraph:RGBColor"/>
    </children>
    <children xsi:type="gmfgraph:RoundedRectangle" name="ESBodyFigure">
      <backgroundColor xsi:type="gmfgraph:RGBColor"
          red="255" green="255" blue="255"/>
    </children>
    <backgroundColor xsi:type="gmfgraph:RGBColor"
        red="128" green="255" blue="128"/>
    <preferredSize dx="100" dy="55"/>
  </figures>
  <figures xsi:type="gmfgraph:RoundedRectangle"
      referencingElements="SessionManager" name="SessionManagerFigure">
    <children xsi:type="gmfgraph:Label" name="SessionManagerLabelFigure"
        text="Session Manager">
      <foregroundColor xsi:type="gmfgraph:RGBColor"/>
    </children>
    <children xsi:type="gmfgraph:RoundedRectangle"
        name="SessionManagerBodyFigure">
      <layout xsi:type="gmfgraph:FlowLayout" vertical="true"
          forceSingleLine="true" majorSpacing="0" minorSpacing="0"/>
      <children xsi:type="gmfgraph:Label"
          name="SessionManagerNameLabelFigure" text="Name">
        <insets left="3"/>
      </children>
      <children xsi:type="gmfgraph:Label"
          referencingElements="SessionManagerSessionname"
          name="SessionManagerNameTextFigure" text="&lt;...>">
        <insets left="10"/>
      </children>
      <backgroundColor xsi:type="gmfgraph:RGBColor"
          red="255" green="255" blue="255"/>
    </children>
    <backgroundColor xsi:type="gmfgraph:RGBColor"
        red="125" green="125" blue="255"/>
    <preferredSize dx="200" dy="55"/>
  </figures>
  <figures xsi:type="gmfgraph:RoundedRectangle"
      referencingElements="MergeManager" name="MergeManagerFigure">
    <children xsi:type="gmfgraph:Label" name="MergeManagerLabelFigure"
        text="Merge Manager">
      <foregroundColor xsi:type="gmfgraph:RGBColor"/>
    </children>
    <children xsi:type="gmfgraph:RoundedRectangle"
        name="MergeManagerBodyFigure">
      <layout xsi:type="gmfgraph:FlowLayout" vertical="true"
          forceSingleLine="true" majorSpacing="0" minorSpacing="0"/>
      <children xsi:type="gmfgraph:Label"
          name="MergeManagerNameLabelFigure" text="Name">
        <insets left="3"/>
      </children>
      <children xsi:type="gmfgraph:Label"
          referencingElements="MergeManagerMergename"
          name="MergeManagerNameTextFigure" text="&lt;...>">
        <insets left="10"/>
      </children>
      <backgroundColor xsi:type="gmfgraph:RGBColor"
          red="255" green="255" blue="255"/>
    </children>
    <backgroundColor xsi:type="gmfgraph:RGBColor"
        red="255" green="115" blue="115"/>
    <preferredSize dx="200" dy="55"/>
  </figures>
  <figures xsi:type="gmfgraph:RoundedRectangle"
      referencingElements="CorrelationManager"
      name="CorrelationManagerFigure">
    <children xsi:type="gmfgraph:Label"
        name="CorrelationManagerLabelFigure" text="Correlation Manager">
      <foregroundColor xsi:type="gmfgraph:RGBColor"/>
    </children>
    <children xsi:type="gmfgraph:RoundedRectangle"
        name="CorrelationManagerBodyFigure">
      <layout xsi:type="gmfgraph:FlowLayout" vertical="true"
          forceSingleLine="true" majorSpacing="0" minorSpacing="0"/>
      <children xsi:type="gmfgraph:Label"
          name="CorrelationManagerNameLabelFigure" text="Name">
        <insets left="3"/>
      </children>
      <children xsi:type="gmfgraph:Label"
          referencingElements="CorrelationManagerName"
          name="CorrelationManagerNameTextFigure" text="&lt;...>">
        <insets left="10"/>
      </children>
      <backgroundColor xsi:type="gmfgraph:RGBColor"
          red="255" green="255" blue="255"/>
    </children>
    <backgroundColor xsi:type="gmfgraph:RGBColor"
        red="192" green="192" blue="192"/>
    <preferredSize dx="200" dy="55"/>
  </figures>
  <figures xsi:type="gmfgraph:RoundedRectangle"
      referencingElements="TupleCorrelation"
      name="TupleCorrelationFigure">
    <children xsi:type="gmfgraph:Label"
        name="TupleCorrelationNameFigure"
        text="Tuple Correlation">
      <foregroundColor xsi:type="gmfgraph:RGBColor"/>
    </children>
    <children xsi:type="gmfgraph:RoundedRectangle"
        name="TupleCorrelationBodyFigure">
      <layout xsi:type="gmfgraph:FlowLayout" vertical="true"
          forceSingleLine="true" majorSpacing="0" minorSpacing="0"/>
      <children xsi:type="gmfgraph:Label"
          name="TupleCorrelationIdentifierLabelFigure"
          text="Identifier">
        <insets left="3"/>
      </children>
      <children xsi:type="gmfgraph:Label"
          referencingElements="TupleCorrelationIdentifier"
          name="TupleCorrelationIdentifierTextFigure" text="&lt;...>">
        <insets left="10"/>
      </children>
      <backgroundColor xsi:type="gmfgraph:RGBColor"
          red="255" green="255" blue="255"/>
    </children>
    <backgroundColor xsi:type="gmfgraph:RGBColor"
        red="255" green="175" blue="60"/>
    <preferredSize dx="200" dy="55"/>
  </figures>
  <figures xsi:type="gmfgraph:PolylineConnection"
      referencingElements="AConnection" name="Connection"
      targetDecoration="ClosedArrowFigure"/>
  <figures xsi:type="gmfgraph:PolylineDecoration"
      name="ClosedArrowFigure">
    <template/>
    <template x="-2" y="2"/>
    <template x="-2" y="-2"/>
    <template/>
  </figures>
  <figures xsi:type="gmfgraph:PolylineConnection"
      referencingElements="ARelation" name="Relation"
      lineKind="LINE_DASH" targetDecoration="OpenArrowFigure">
    <children xsi:type="gmfgraph:PolylineDecoration"
        name="OpenArrowFigure">
      <template x="-1" y="1"/>
      <template/>
      <template x="-1" y="-1"/>
    </children>
  </figures>
</figures>
<nodes name="ES" figure="ESFigure"/>
<nodes name="SessionManager" figure="SessionManagerFigure"/>
<nodes name="MergeManager" figure="MergeManagerFigure"/>
<nodes name="CorrelationManager" figure="CorrelationManagerFigure"/>
<nodes name="TupleCorrelation" figure="TupleCorrelationFigure"/>
<connections name="AConnection" figure="Connection"/>
<connections name="ARelation" figure="Relation"/>
<labels name="SessionManagerSessionname"
    figure="SessionManagerNameTextFigure" elementIcon="false"/>
<labels name="MergeManagerMergename"
    figure="MergeManagerNameTextFigure" elementIcon="false"/>
<labels name="CorrelationManagerName"
    figure="CorrelationManagerNameTextFigure" elementIcon="false"/>
<labels name="TupleCorrelationIdentifier"
    figure="TupleCorrelationIdentifierTextFigure" elementIcon="false"/>
</gmfgraph:Canvas>
```

A.3 EventServer.gmftool

```
<?xml version="1.0" encoding="UTF-8"?>
<gmftool:ToolRegistry xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:gmftool="http://www.eclipse.org/gmf/2005/ToolDefinition">
  <palette>
    <tools xsi:type="gmftool:ToolGroup" title="Nodes">
      <tools xsi:type="gmftool:CreationTool" title="ES"
          description="Create new ES">
        <smallIcon xsi:type="gmftool:DefaultImage"/>
        <largeIcon xsi:type="gmftool:DefaultImage"/>
      </tools>
      <tools xsi:type="gmftool:CreationTool" title="SessionManager"
          description="Create new SessionManager">
        <smallIcon xsi:type="gmftool:DefaultImage"/>
        <largeIcon xsi:type="gmftool:DefaultImage"/>
      </tools>
      <tools
          xsi:type="gmftool:CreationTool" title="MergeManager"
          description="Create new MergeManager">
        <smallIcon xsi:type="gmftool:DefaultImage"/>
        <largeIcon xsi:type="gmftool:DefaultImage"/>
      </tools>
      <tools xsi:type="gmftool:CreationTool" title="CorrelationManager"
          description="Create new CorrelationManager">
        <smallIcon xsi:type="gmftool:DefaultImage"/>
        <largeIcon xsi:type="gmftool:DefaultImage"/>
      </tools>
      <tools xsi:type="gmftool:CreationTool" title="TupleCorrelation"
          description="Create new TupleCorrelation">
        <smallIcon xsi:type="gmftool:DefaultImage"/>
        <largeIcon xsi:type="gmftool:DefaultImage"/>
      </tools>
    </tools>
    <tools xsi:type="gmftool:ToolGroup" title="Links">
      <tools xsi:type="gmftool:CreationTool"
          title="Connection" description="Create new Connection">
        <smallIcon xsi:type="gmftool:DefaultImage"/>
        <largeIcon xsi:type="gmftool:DefaultImage"/>
      </tools>
      <tools xsi:type="gmftool:CreationTool"
          title="Relation" description="Create new Relation">
        <smallIcon xsi:type="gmftool:DefaultImage"/>
        <largeIcon xsi:type="gmftool:DefaultImage"/>
      </tools>
    </tools>
  </palette>
</gmftool:ToolRegistry>
```

A.4 EventServer.gmfmap

```
<?xml version="1.0" encoding="UTF-8"?>
<gmfmap:Mapping xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
    xmlns:gmfmap="http://www.eclipse.org/gmf/2005/mappings"
    xmlns:gmftool="http://www.eclipse.org/gmf/2005/ToolDefinition">
  <nodes>
    <containmentFeature href="EventServer.ecore#//Diagram/mergemanager"/>
    <ownedChild>
      <domainMetaElement href="EventServer.ecore#//MergeManager"/>
      <labelMappings editPattern="">
        <diagramLabel href="EventServer.gmfgraph#MergeManagerMergename"/>
        <features href="EventServer.ecore#//MergeManager/mergename"/>
      </labelMappings>
      <tool xsi:type="gmftool:CreationTool"
          href="EventServer.gmftool#//@palette/@tools.0/@tools.2"/>
      <diagramNode href="EventServer.gmfgraph#MergeManager"/>
    </ownedChild>
  </nodes>
  <nodes>
    <containmentFeature href="EventServer.ecore#//Diagram/sessionmanager"/>
    <ownedChild>
      <domainMetaElement href="EventServer.ecore#//SessionManager"/>
      <labelMappings editPattern="">
        <diagramLabel href="EventServer.gmfgraph#SessionManagerSessionname"/>
        <features href="EventServer.ecore#//SessionManager/sessionname"/>
      </labelMappings>
      <tool xsi:type="gmftool:CreationTool"
          href="EventServer.gmftool#//@palette/@tools.0/@tools.1"/>
      <diagramNode href="EventServer.gmfgraph#SessionManager"/>
    </ownedChild>
  </nodes>
  <nodes>
    <containmentFeature href="EventServer.ecore#//Diagram/eventserver"/>
    <ownedChild>
      <domainMetaElement href="EventServer.ecore#//ES"/>
      <tool xsi:type="gmftool:CreationTool"
          href="EventServer.gmftool#//@palette/@tools.0/@tools.0"/>
      <diagramNode href="EventServer.gmfgraph#ES"/>
    </ownedChild>
  </nodes>
  <nodes>
    <containmentFeature href="EventServer.ecore#//Diagram/tuplecorrelation"/>
    <ownedChild>
      <domainMetaElement href="EventServer.ecore#//TupleCorrelation"/>
      <labelMappings editPattern="">
        <diagramLabel href="EventServer.gmfgraph#TupleCorrelationIdentifier"/>
        <features href="EventServer.ecore#//TupleCorrelation/identifier"/>
      </labelMappings>
      <tool xsi:type="gmftool:CreationTool"
          href="EventServer.gmftool#//@palette/@tools.0/@tools.4"/>
      <diagramNode href="EventServer.gmfgraph#TupleCorrelation"/>
    </ownedChild>
  </nodes>
  <nodes>
    <containmentFeature href="EventServer.ecore#//Diagram/correlationmanager"/>
    <ownedChild>
      <domainMetaElement href="EventServer.ecore#//CorrelationManager"/>
      <labelMappings editPattern="">
        <diagramLabel href="EventServer.gmfgraph#CorrelationManagerName"/>
        <features href="EventServer.ecore#//CorrelationManager/name"/>
      </labelMappings>
      <tool xsi:type="gmftool:CreationTool"
          href="EventServer.gmftool#//@palette/@tools.0/@tools.3"/>
      <diagramNode href="EventServer.gmfgraph#CorrelationManager"/>
    </ownedChild>
  </nodes>
  <links>
    <domainMetaElement
        href="EventServer.ecore#//CorrelationManagerTupleCorrelationConnection"/>
    <containmentFeature href="EventServer.ecore#//CorrelationManager/
        correlationmanagertuplecorrelationconnection"/>
    <tool xsi:type="gmftool:CreationTool"
        href="EventServer.gmftool#//@palette/@tools.1/@tools.0"/>
    <diagramLink href="EventServer.gmfgraph#AConnection"/>
    <sourceMetaFeature xsi:type="ecore:EReference" href="EventServer.ecore#
        //CorrelationManagerTupleCorrelationConnection/source"/>
    <linkMetaFeature xsi:type="ecore:EReference" href="EventServer.ecore#
        //CorrelationManagerTupleCorrelationConnection/target"/>
  </links>
  <links>
    <domainMetaElement
        href="EventServer.ecore#//EventServerCorrelationManagerConnection"/>
    <containmentFeature
```

```
        href="EventServer.ecore#//ES/eventservercorrelationmanagerconnection"/>
    <tool xsi:type="gmftool:CreationTool"
        href="EventServer.gmftool#//@palette/@tools.1/@tools.0"/>
    <diagramLink href="EventServer.gmfgraph#AConnection"/>
    <sourceMetaFeature xsi:type="ecore:EReference"
        href="EventServer.ecore#//EventServerCorrelationManagerConnection/source"/>
    <linkMetaFeature xsi:type="ecore:EReference"
        href="EventServer.ecore#//EventServerCorrelationManagerConnection/target"/>
  </links>
  <links>
    <domainMetaElement
        href="EventServer.ecore#//EventServerSessionManagerConnection"/>
    <containmentFeature
        href="EventServer.ecore#//ES/eventserversessionmanagerconnection"/>
    <tool xsi:type="gmftool:CreationTool"
        href="EventServer.gmftool#//@palette/@tools.1/@tools.0"/>
    <diagramLink href="EventServer.gmfgraph#AConnection"/>
    <sourceMetaFeature xsi:type="ecore:EReference"
        href="EventServer.ecore#//EventServerSessionManagerConnection/source"/>
    <linkMetaFeature xsi:type="ecore:EReference"
        href="EventServer.ecore#//EventServerSessionManagerConnection/target"/>
  </links>
  <links>
    <domainMetaElement
        href="EventServer.ecore#//SessionManagerMergeManagerConnection"/>
    <containmentFeature href="EventServer.ecore#
        //SessionManager/sessionmanagermergemanagerconnection"/>
    <tool xsi:type="gmftool:CreationTool"
        href="EventServer.gmftool#//@palette/@tools.1/@tools.0"/>
    <diagramLink href="EventServer.gmfgraph#AConnection"/>
    <sourceMetaFeature xsi:type="ecore:EReference"
        href="EventServer.ecore#//SessionManagerMergeManagerConnection/source"/>
    <linkMetaFeature xsi:type="ecore:EReference"
        href="EventServer.ecore#//SessionManagerMergeManagerConnection/target"/>
  </links>
  <links>
    <domainMetaElement href="EventServer.ecore#
        //SessionManagerCorrelationManagerRelation"/>
    <containmentFeature href="EventServer.ecore#
        //SessionManager/sessionmanagercorrelationmanagerrelation"/>
    <tool xsi:type="gmftool:CreationTool"
        href="EventServer.gmftool#//@palette/@tools.1/@tools.1"/>
    <diagramLink href="EventServer.gmfgraph#ARelation"/>
    <sourceMetaFeature xsi:type="ecore:EReference" href="EventServer.ecore#
        //SessionManagerCorrelationManagerRelation/source"/>
    <linkMetaFeature xsi:type="ecore:EReference" href="EventServer.ecore#
        //SessionManagerCorrelationManagerRelation/target"/>
  </links>
  <diagram>
    <diagramCanvas href="EventServer.gmfgraph#EventServer"/>
    <domainModel href="EventServer.ecore#/"/>
    <domainMetaElement href="EventServer.ecore#//Diagram"/>
    <palette href="EventServer.gmftool#//@palette"/>
  </diagram>
</gmfmap:Mapping>
```

A.5 workflow.oaw

```
<workflow>
  <property file='workflow.properties'/>
  <component id="xmiParser" class="org.openarchitectureware.emf.XmiReader">
      <modelFile value="${modelFile}"/>
      <metaModelPackage value="EventServer.EventServerPackage"/>
      <outputSlot value="model"/>
      <firstElementOnly value="true"/>
    </component>
  <component id="dirCleaner" class="oaw.workflow.common.DirectoryCleaner"
      directories="gen"/>
  <component id="generator" class="oaw.xpand2.Generator" skipOnErrors="true">
    <metaModel class="oaw.type.emf.EmfMetaModel">
      <metaModelPackage value="EventServer.EventServerPackage"/>
    </metaModel>
    <expand value="src::mapping::diagram FOR model"/>
    <genPath value="${srcGenPath}/"/>
    <srcPath value="${srcGenPath}/"/>
  </component>
</workflow>
```

A.6 workflow.properties

```
modelFile=ES1.eventserver
srcGenPath=gen
```

A.7 mapping.xpt

```
Â«DEFINE diagram FOR EventServer::Diagram-Â»
Â«FILE "Config.xml"-Â»
    <?xml version="1.0" encoding="UTF-8"?>
    Â«EXPAND es FOR eventserver-Â»
Â«ENDFILE-Â»
Â«ENDDEFINEÂ»
Â«DEFINE es FOR EventServer::ES-Â»
  <eventserver>
    Â«EXPAND cm FOREACH eventservercorrelationmanagerconnection-Â»
    Â«EXPAND esr FOREACH eventserversessionmanagerconnection-Â»
  </eventserver>
Â«ENDDEFINEÂ»
Â«DEFINE esr FOR EventServer::EventServerSessionManagerConnection-Â»
  Â«IF target!=null-Â»
    Â«EXPAND s FOR target-Â»
  Â«ENDIF-Â»
Â«ENDDEFINEÂ»
Â«DEFINE s FOR EventServer::SessionManager-Â»
  <sessionmanager name="Â«sessionnameÂ»" correlationManager="Â«EXPAND scr FOR
      sessionmanagercorrelationmanagerrelation-Â»">
    Â«EXPAND smr FOR sessionmanagermergemanagerconnection-Â»
  <sessionmanager/>
Â«ENDDEFINEÂ»
Â«DEFINE scr FOR EventServer::SessionManagerCorrelationManagerRelation-Â»
```

```
  Â«target.name-Â»
Â«ENDDEFINEÂ»
Â«DEFINE smr FOR EventServer::SessionManagerMergeManagerConnection-Â»
  <mergeManager name="Â«target.mergenameÂ»"/>
Â«ENDDEFINEÂ»
Â«DEFINE cm FOR EventServer::EventServerCorrelationManagerConnection-Â»
  Â«IF target!=null-Â»
    Â«EXPAND t FOR target-Â»
  Â«ENDIF-Â»
Â«ENDDEFINEÂ»
Â«DEFINE t FOR EventServer::CorrelationManager-Â»
  <correlationManager name="Â«nameÂ»">
    Â«EXPAND ctr FOREACH correlationmanagertuplecorrelationconnection-Â»
  </correlationManager>
Â«ENDDEFINEÂ»
Â«DEFINE ctr FOR EventServer::CorrelationManagerTupleCorrelationConnection-Â»
  Â«IF target!=null-Â»
    Â«EXPAND ct FOR target-Â»
  Â«ENDIF-Â»
Â«ENDDEFINEÂ»
Â«DEFINE ct FOR EventServer::TupleCorrelation-Â»
  <tupleCorrelation identifier="Â«identifierÂ»"/>
Â«ENDDEFINEÂ»
```

A.8 NewAction.java

```
package eventserver.jdom.popup.actions;
import java.io.File;
import java.io.FileOutputStream;
import java.util.List;
import org.eclipse.core.resources.IFile;
import org.eclipse.jface.action.IAction;
import org.eclipse.jface.dialogs.MessageDialog;
import org.eclipse.jface.viewers.ISelection;
import org.eclipse.jface.viewers.IStructuredSelection;
import org.eclipse.swt.widgets.Shell;
import org.eclipse.ui.IObjectActionDelegate;
import org.eclipse.ui.IWorkbenchPart;
import org.jdom.Attribute;
import org.jdom.Document;
import org.jdom.Element;
import org.jdom.input.SAXBuilder;
import org.jdom.output.Format;
import org.jdom.output.XMLOutputter;

public class NewAction implements IObjectActionDelegate {
IFile file;
public NewAction() {
super();
}
public void setActivePart(IAction action, IWorkbenchPart targetPart) {
}
public void run(IAction action) {
Shell shell = new Shell();
try {
Document D = buildDomDocument();
Element DRoot=D.getRootElement();
Document E= createEmptyDomDocument();
addAllElements(E,DRoot);
saveDocument(E);
}
catch (Exception e) {
MessageDialog.openInformation(shell,"Exception!!!",e.getMessage());
}

MessageDialog.openInformation(
shell,"MuleEditorDOM Plug-in","Create Config... was executed.");
}
public Document buildDomDocument() throws Exception {
SAXBuilder builder = new SAXBuilder();
    Document D = builder.build(file.getLocation().toFile().getAbsolutePath());
return D;
}
  public Document createEmptyDomDocument() throws Exception {
Document doc=new Document();
return doc;
}
public void saveDocument(Document E) throws Exception {
String fname=file.getLocation().toString();
int ind=fname.lastIndexOf("/");
fname=fname.substring(0, ind+1);
File f=new File(fname+"output.xml");
FileOutputStream fo=new FileOutputStream(f);
    XMLOutputter out = new XMLOutputter( Format.getPrettyFormat() );
  out.output( E,fo);
}
public void addAllElements(Document E,Element R) throws Exception {
Element root=createElement(R.getChild("eventserver"));
Element es=R.getChild("eventserver");
List rootchilds=es.getChildren();
for(int i=0;i<rootchilds.size();i++) {
String targetVal=((Element)rootchilds.get(i)).getAttributeValue("target");
System.out.println(targetVal);
if(targetVal !=null) {
addElement(R,root,es,targetVal);
}
}
E.setRootElement(root);
}
public void addElement(Element root,Element addEl, Element deep,String ref) throws Exception {
Element Childref=getChildrenRef(root,deep,ref);
Element item=createElement(Childref);
List childs=Childref.getChildren();
for(int j=0;j<childs.size();j++) {
String targetVal=((Element)childs.get(j)).getAttributeValue("target");
String targetName=((Element)childs.get(j)).getName();
if(targetName.endsWith("relation")) {
Element att=getChildrenRef(root,item,targetVal);
item.setAttribute(att.getName(),att.getAttributeValue("name"));
}else {
if(targetVal !=null) {
addElement(root,item,Childref,targetVal);
}
}
}
addEl.addContent(item);
}
public Element createElement(Element toAdd) throws Exception {
Element A=new Element(toAdd.getName());
List Attribs=toAdd.getAttributes();
```

```
for(int i=0;i<Attribs.size();i++) {
Attribute a=(Attribute)Attribs.get(i);
A.setAttribute(a.getName(),a.getValue());
}
return A;
}
public Element getChildrenRef(Element Root,Element E,String reference) throws Exception{
int ind;
String TagName;
try {ind=Integer.parseInt(reference.substring(reference.length()-1));}
catch(Exception e) {ind=-1;}
int indat=reference.indexOf("@");
if(ind==-1) TagName=reference.substring(indat+1,reference.length());
else TagName=reference.substring(indat+1,reference.length()-2);
System.out.println("TagName: "+TagName);
List sessions=Root.getChildren(TagName);
if(ind==-1) return (Element)sessions.get(0);
else return (Element)sessions.get(ind);
}
public void selectionChanged(IAction action, ISelection selection) {
file = (IFile) ((IStructuredSelection) selection).getFirstElement();
}
}
```

A.9 eventserver.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<eventServer xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="eventserver.xsd">
  <correlationManager name="eventServerManager">
    <correlationBridge identifier="bridge">
      <correlationSet identifier="flightProcess">
        <correlationItem identifier="flight"
            eventType="eventobjecttype://www.examples.com/FlightPlan/FlightPlan">
          <xPathSelector>//aircraftID</xPathSelector>
        </correlationItem>
        <correlationItem identifier="arrival"
            eventType="eventobjecttype://www.examples.com/FlightPlan/Arrival">
          <xPathSelector>//aircraftID</xPathSelector>
        </correlationItem>
        <correlationItem identifier="departure"
            eventType="eventobjecttype://www.examples.com/FlightPlan/Departure">
          <xPathSelector>//aircraftID</xPathSelector>
        </correlationItem>
        <correlationItem identifier="trackFlightPlan"
            eventType="eventobjecttype://www.examples.com/FlightPlan/TrackFlightPlan">
          <xPathSelector>//aircraftID</xPathSelector>
        </correlationItem>
      </correlationSet>
      <correlationSet identifier="track">
        <correlationItem identifier="trackFlightPlan"
            eventType="eventobjecttype://www.examples.com/FlightPlan/TrackFlightPlan">
          <xPathSelector>//AtmsFlightPlanID</xPathSelector>
        </correlationItem>
        <correlationItem identifier="trackFlightPlan"
            eventType="eventobjecttype://www.examples.com/FlightPlan/TrackMessage">
          <xPathSelector>//AtmsFlightPlanID</xPathSelector>
        </correlationItem>
        <correlationItem identifier="trackFlightPlan"
            eventType="eventobjecttype://www.examples.com/FlightPlan/TrackCancelMessage">
          <xPathSelector>//AtmsFlightPlanID</xPathSelector>
        </correlationItem>
      </correlationSet>
    </correlationBridge>
  </correlationManager>
  <sessionManager name="eventServerManager"
    class="at.ac.tuwien.ifs.eventserver.core.session.SessionManagerImpl"
    correlationManager="eventServerManager">
  </sessionManager>
</eventServer>
```

A.10 eventserver-mule-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<mule-configuration id="Event_Server" version="1.0">
  <description>
    This mule configuration file configures all necessary umo,
    transformer and endpoints to be used by the event server
  </description>
  <agents>
    <agent name="eventServerManager"
      className="at.ac.tuwien.ifs.eventserver.core.EventServerManager"/>
  </agents>
  <connector name="jmsConnector" className="org.mule.providers.jms.JmsConnector">
    <properties>
      <property name="specification" value="1.1"/>
      <property name="connectionFactoryJndiName" value="ConnectionFactory"/>
      <property name="jndiInitialFactory"
        value="org.activemq.jndi.ActiveMQInitialContextFactory"/>
      <map name="connectionFactoryProperties">
        <property name="brokerURL" value="vm://localhost"/>
        <property name="brokerXmlConfig" value="classpath:activemq.xml"/>
      </map>
    </properties>
  </connector>
  <transformers>
    <transformer name="ActiveMQToSenactiveEvent"
      className="at.ac.tuwien.ifs.eventserver.core.transformer.ActiveMQToSenactiveEvent"
      returnClass="at.ac.tuwien.ifs.eventserver.core.event.SenactiveEvent"/>
  </transformers>
  <model name="eventServer">
    <mule-descriptor name="UuidUmo"
      implementation="at.ac.tuwien.ifs.eventserver.core.umo.UuidUmo">
      <inbound-router>
        <endpoint address="jms://MyDestination" transformers="ActiveMQToSenactiveEvent"/>
      </inbound-router>
      <outbound-router>
        <router className="org.mule.routing.outbound.OutboundPassThroughRouter">
          <endpoint address="vm://uuid"/>
        </router>
      </outbound-router>
    </mule-descriptor>
    <mule-descriptor name="CorrelationUmo"
      implementation="at.ac.tuwien.ifs.eventserver.core.umo.CorrelationUmo">
      <inbound-router>
        <endpoint address="vm://uuid"/>
      </inbound-router>
      <outbound-router>
        <router className="org.mule.routing.outbound.OutboundPassThroughRouter">
          <endpoint address="vm://correlated"/>
        </router>
      </outbound-router>
      <properties>
        <property name="correlationManagerId" value="correlations"/>
      </properties>
    </mule-descriptor>
    <mule-descriptor name="PersistFlightUmo" implementation=" at.ac.tuwien.ifs.
        eventserver.examples.flightplan.umo.PersistFlightUmo">
      <inbound-router>
        <endpoint address="vm://correlated"/>
      </inbound-router>
      <outbound-router>
        <router className="org.mule.routing.outbound.OutboundPassThroughRouter">
          <endpoint address="vm://persisted"/>
        </router>
      </outbound-router>
      <properties>
        <property name="sessionManagerId" value="sessions"/>
      </properties>
    </mule-descriptor>
    <mule-descriptor name="CalculateDurationUmo" implementation=" at.ac.tuwien.ifs.
        eventserver.examples.flightplan.umo.CalculateDurationUmo">
      <inbound-router>
        <endpoint address="vm://correlated"/>
      </inbound-router>
      <outbound-router>
        <router className="org.mule.routing.outbound.OutboundPassThroughRouter">
          <endpoint address="vm://calculated"/>
        </router>
      </outbound-router>
      <properties>
        <property name="sessionManagerId" value="sessions"/>
      </properties>
    </mule-descriptor>
    <mule-descriptor name="PrintEventUmo"
        implementation="at.ac.tuwien.ifs.eventserver.core.umo.PrintEventUmo">
      <inbound-router>
        <endpoint address="vm://calculated"/>
      </inbound-router>
      <properties>
        <property name="sessionManagerId" value="sessions"/>
      </properties>
    </mule-descriptor>
  </model>
</mule-configuration>
```

List of Tables

List of Figures

Bibliography

[BERCZUK 2003] BERCZUK, STEPHEN (2003). *Software Configuration Management Patterns, Effective Teamwork, Practical Integration*. Addison-Wesley.

[BERGER 2006] BERGER, RONALD (2006). *IFS Eventserver Overview*.

[BERSOFF et al. 1978] BERSOFF, EDWARD H., V. D. HENDERSON and S. G. SIEGEL (1978). *Software Configuration Management*.

[CAPADOUCA 2004] CAPADOUCA, CHRIS (2004). *Model Driven Configuration Management*.

[CENTER 2005] CENTER, SOFTWARE TECHNOLOGY SUPPORT (2005). *Configuration Management Fundamentals*. Technical Report, Software Technology Support Center.

[DART 1994] DART, SUSAN (1994). *Concepts in Configuration Management Systems*. Technical Report, Software Engineering Institute, Carnegie-Mellon University.

[EFFTINGE] EFFTINGE, SVEN. *OpenArchitectureWare 4.1 Check Validation Language*. http://www.eclipse.org/gmt/oaw/doc/4.1/r10_checkReference.pdf.

[EFFTINGE and VOELTER] EFFTINGE, SVEN and M. VOELTER. *OpenArchitectureWare 4.1 Workflow Engine Reference*. http://www.eclipse.org/gmt/oaw/doc/4.1/r05_workflowReference.pdf.

[EHRIG et al. 2005] EHRIG, KARSTEN, C. ERMEL, S. HANSGEN and G. TAENTZER (2005). *Generation of Visual Editors as Eclipse PlugIns*. Technical Report, Technical University of Berlin, Germany.

[ESTUBLIER 2000] ESTUBLIER, JACKY (2000). *Software Configuration Management A Roadmap*.

[FRANCE and RUMPE 2007] FRANCE, ROBERT and B. RUMPE (2007). *Model-driven Development of Complex Software: A Research Roadmap*. Technical Report, Colorado State University/Braunschweig University of Technology.

[FRANKEL 2003] FRANKEL, DAVID S. (2003). *Model Driven Architecture*. John Wiley & Sons.

[GRINTER 1996] GRINTER, REBECCA E. (1996). *Using a Configuration Management Tool to Coordinate Software Development*. Technical Report, Department of Information and Computer Science, University of California, Irvine.

[HOU et al. 2001] HOU, JINGYU, Y. ZHANG and Y. KAMBAYASHI (2001). *Object-Oriented Representation for XML Data*. Technical Report, Dept. of Math. and Comput., Univ. of Southern Queensland.

[IVAN KURTEV 2003] IVAN KURTEV, KLAAS VAN DEN BERG (2003). *Model Driven Architecture based XML Processing*. Technical Report, Software Engineering Group, University of Twente.

[JACKSON and RINARD 2000] JACKSON, DANIEL and M. RINARD (2000). *Software Analysis: A Roadmap*.

[JACKY ESTUBLIER 2005] JACKY ESTUBLIER, GEOFFREY CLEMM, DAVID LEBLANG (2005). *Impact of Software Engineering Research on the Practice of Software Configuration Management*. Technical Report, Impact Project.

[JECKLE et al. 2003] JECKLE, MARIO, C. RUPP, J. HAHN, B. ZENGLER and S. QUEINS (2003). *UML 2 glasklar*. Hanser Fachbuchverlag.

[JOERIS 1998] JOERIS, GREGOR (1998). *Change Management Needs Integrated Process and Configuration Management*. Technical Report, Intelligent Systems Department, University of Bremen.

[KOEGEL 2008] KOEGEL, MAXIMILIAN (2008). *Towards Software Configuration Management for Unified Models*. Technical Report, Department of Computer Science, Technische Universitaet Muenchen.

[LEON 2005] LEON, ALEXIS (2005). *Software Configuration Management Handbook, 2nd Edition*. Artech House.

[LIPIEN et al. 2006] LIPIEN, DAVID, J. HAINES and P. GAN (2006). *Enterprise Software Release Management*. Technical Report, IBM.

[MEHROTRA 2004] MEHROTRA, VIB (2004). *Release Management CMJournal*. CM Journal.

[MICROSOFT 2004] MICROSOFT (2004). *Release Management TechNet*. Technical Report, Microsoft.

[MOREIRA 2005] MOREIRA, MARIO (2005). *ABCs of Requirements Engineering*. CM Journal.

[PETRASCH and MEIMBERG 2006] PETRASCH, ROLAND and O. MEIMBERG (2006). *Model Driven Architecture - Eine praxisorientierte Einfuehrung in die MDA (German)*. dpunkt.

[POPP 2008] POPP, GUNTHER (2008). *Konfigurationsmanagement mit Subversion, Ant und Maven*. dpunkt.

[QUATRANI and PALISTRANT 2006] QUATRANI, TERRY and J. PALISTRANT (2006). *Visual Modeling with IBM Rational Software Architect and UML*. IBM Press.

[RENDER and CAMPBELL 1991] RENDER, HAL and R. CAMPBELL (1991). *An Object-Oriented Model of Software Configuration Management*. Technical Report, Department of Computer Science; University of Colorado, University of Illinois.

[REUSSNER and BECKER 2005] REUSSNER, RALF H. and S. BECKER (2005). *Testplan*. Technical Report, Software Engineering Universitaet Oldenburg.

[ROUTLEDGE et al. 2001] ROUTLEDGE, NICHOLAS, L. BIRD and A. GOODCHILD (2001). *UML and XML Schema*. Technical Report, Distributed Systems Technology Centre, University of Queensland, Australia.

[ROYCE 1987] ROYCE, W.W. (1987). *Managing the development of large software systems*.

[SALIM et al. 2004] SALIM, FLORA DILYS, R. PRICE and S. KRISHNASWAMY (2004). *UML Documentation Support for XML Schema*. Technical Report, School of Computer Science and Software Engineering,Business Systems Monash University, Victoria, Australia.

[SAURER 2006] SAURER, GERALD (2006). *Flightplan Tutorial*.

[SHATALIN and TIKHOMIROV 2006] SHATALIN, ALEXANDER and A. TIKHOMIROV (2006). *Graphical Modeling Framework Architecture Overview*.

[STAHL and VOELTER 2006] STAHL, TOM and M. VOELTER (2006). *Model-Driven Software Development - Technology, Engineering, Management*. Wiley.

[VOELTER 2007] VOELTER, MARCUS (2007). *openArchitectureWare 4 - the flexible open source tool platform for model-driven software development.*.

[VOELTER] VOELTER, MARKUS. *OpenArchitectureWare 4.1 EMF Example*. http://www.eclipse.org/gmt/oaw/doc/4.1/30_emfExample.pdf.

[VOELTER 2003] VOELTER, MARKUS (2003). *Metamodellierung (German)*.

[WEISMANN 2006] WEISMANN, BENEDIKT (2006). *Architekturzentrierte Modellgetriebene Softwareentwicklung - Fallbeispiel und Evaluierung (German)*. PhD thesis, Technischen Universitaet Wien.

[ZEPPENFELD and WOLTERS 2006] ZEPPENFELD, KLAUS and R. WOLTERS (2006). *Generative Software-Entwicklung mit der MDA (German)*. Spektrum.

[ZUSER et al. 2001] ZUSER, WOLFGANG, S. BIFFL, T. GRECHENIG and M. KOEHLE (2001). *Software Engineering mit UML und dem Unified Process*. Pearson Studium.