Christian-Albrechts-University Kiel

Department of Computer Science

Software Engineering Group

**Bachelorthesis**

# Modeling Usage and Architecture Metrics for Software Systems Applying OMG's KDM and SMM

Benjamin Schnoor

September 30, 2010

Supervised by: Prof. Dr. Wilhelm Hasselbring

M.Sc. Sören Frey

## Declaration of Authorship

I Benjamin Schnoor hereby declare that this thesis and the work presented in it is entirely my own. Where I have consulted the work of others, this is always clearly stated.

Kiel, 30. September 2010

________________________

(Benjamin Schnoor)

Abstract

In 1994, David Parnas remarked that most software systems age over time [18]. He proposes reengineering in order to slow down software aging. Reengineering consists of two phases. First, the software system must be analyzed to get different representations of the existing system. Afterwards, the software system can be restructured in order to improve internal software quality. The Object Management Group founded the Architecture-Driven Modernization Task Force which should develop a set of modernization standards. These standards aim to support the reengineering process throughout both phases. The corresponding Software Metrics Meta-Model (SMM) and the Knowledge Discovery Meta-Model (KDM) can be utilized in order to apply software metrics to software systems. KDM is used to represent different views on software systems while SMM is used to model software metrics. According to the specification, SMM instances can be applied to KDM instances. However, J.L.C. Izquierdo and J.G. Molina remark that, there does not exist a generic execution engine yet [13]. This thesis describes how SMM can be utilized to model software metrics. We introduce a set of Eclipse plugins which implement such an execution engine and contribute to the CloudMIG platform.

# Contents

# List of Figures

# 1 Introduction

## 1.1 Motivation

Most software systems age over time. That is because of the constant change of the existing systems due to varying requirements and new technologies. Often modifications alter the original software architecture and make it more difficult to maintain the system (e.g., see [9]). At some point, there is no possibility to modify the existing software system with acceptable costs because of the eroded software architecture and a migration to a new target architecture is an option to modernize the system. Before an engineer can start with software modernization, he needs information about the legacy system on which he wants to work. These information measured offer valuable clues to the quality of the software system. Common examples for helpful information are:

- The complexity of the architecture
- The grade of cohesion and coupling of software components
- The depth of the inheritance tree

Another aspect for this work is the deployment of software systems in the cloud computing domain [6]. There are many advantages of deploying applications to cloud environments, such as cloud-based applications often use only as much resources as they need. Cloud environments enable a quick adjustment of resources, for example depending on the current workload [10]. For example, the under-provisioning of server resources can be reduced and availability will be increased. Pricing often depends on resource consumption and can also depend on the point in time when the services will be executed, which means that consuming resources at night will probably be cheaper than consuming resources at daytime. Therefore, resource efficiency and intelligent load distribution becomes very important. However, to support this, it is required to understand the usage of the established system. Important questions could be:

- How often was a service called?
- Which resources were consumed?
- When were the resources consumed?

## 1.2 Goal

The Object Management Group (OMG) founded a task force which concentrates on modernizing legacy systems. Therefore, it suggests the Architecture-Driven Modernization (ADM) process[1]. By this approach, it shall be ensured that users consider the modernization from an analysis and design perspective. This helps to prevent that obsolete concepts will be retained in modern programming languages. ADM provides several standard packages to support the reengineering process from the beginning to the end. One of these standards is the Software Metrics Meta-Model (SMM)[2] which represents measures and their measurements related to any MOF[3] conform element. Another comprehensive is the Knowledge Discovery Meta-Model (KDM)[4] which facilitates the modeling of software systems. In [13], the authors provide a case study, in which they use KDM and SMM to restructure legacy Oracle Forms applications. The authors remark, that SMM can represent the measures and the measurements, but it does not provide an execution engine, which applies the measures to KDM instances.

We show how to utilize OMG's ADM standard SMM to model different software metrics. Since the specification of the Software Metrics Meta-Model is only available as an early version (Beta 1.0), we want to discuss some problems of the current version, and how to avoid resulting trouble. Furthermore, we develop an execution engine, which

[1]http://adm.omg.org, accessed 27-September-2010
[2]http://www.omg.org/spec/SMM/1.0/Beta1, accessed 2010-09-27
[3]http://www.omg.org/spec/MOF/2.0, accessed 2010-09-02
[4]http://www.omg.org/spec/KDM/1.1, accessed 2010-09-27

can apply the defined SMM measures to KDM instances. To the best of our knowledge, there does not exist a generic solution yet. So we show, how to translate the SMM elements into source code. Finally, we provide Eclipse plugins which will be used by the CloudMIG framework [10]. One plugin provides the generic definition of SMM-conform models. This is required because it is unknown which software metrics will be important. Another plugin implements the execution engine and therefore controls the application of defined metrics to KDM instances.

The plugins operate on models which are EMF-conform. EMF[5] is the Eclipse Modeling Framework which provides means for code generation and utilizes the Ecore meta-model. In addition, EMF provides tools and support to produce a set of Java classes which facilitate the usage of model instances.

## 1.3 Structure of the Thesis

After this introduction, the thesis begins with an overview of related work in Section 2. First, we deal with software aging. David Parnas analyzed this issue in 1994 the first time. Then, we explain the terms "Reverse engineering" and "Restructuring". Afterewards, we will consider software systems evolution. Though, we talk about a possible classification of software changes. Finally, we deal with software metrics. We will introduce an evaluation framework, which tries to ensure, that the engineer finds adequate measures. In addition, we will see the IEEE "Software Quality Metrics Methodology" standard, which describes a procedure to apply software quality metrics to software systems. This standard can also be applied to other metrics.

In the next Section 3, we will cover the Architecture-Driven Modernization approach of the Object Management Group. Therefore, we will start with the consideration of all

[5]http://www.eclipse.org/emf/

seven ADM components. We will show, which purpose each component serves. Subsequently, we will have a closer look at the KDM and SMM. We will explain the standard Knowledge Discovery Meta-Model and we will show, how to model different aspects of software systems by using this standard. We will emphasize classes, which are very important for architectural metrics. Afterwards, we will concentrate on the Software Metrics Meta-Model. First, we will introduce the structure of the meta model and we will describe all elements, especially, we will talk about the different kinds of measures and the related measurements. Furthermore, we will see how SMM instances can be linked to other model instances. This section ends with an example. Therefore, we will have a short consideration of the McCabe complexity metric and finally, we will show how to model this metrics with SMM.

In the fourth Section, we will introduce the developed tool. After a general description of the architecture, we will present the two plugins in detail.

In Section 5, we will deal with the SMM execution engine. First, we will examine the current version of design flaws of SMM, which may lead to misunderstandings. Because of these flaws, we had to make some assumptions, which make it possible to implement the execution engine. The most important question for the execution manager is: “How to deal with generic measures?”. Each kind of measure needs its own workflow and has got its own operation characteristics, which must be considered separately.

The sixth Section deals with the evaluation of the implemented tool. Therefore, we start with the definition of three different architecture metrics. After that, we generate KDM instances of a very simple Java program, of the NanoHTTPD webserver[6], and of

[6]http://elonen.iki.fi/code/nanohttpd/, accessed 2010-09-27

the Ibatis JPetstore[7] with the help of the MoDisco Eclipse plugin[8]. Subsequently, we apply the metrics to the KDM models and save the results. In the next step, we apply the same metrics with an established tool for comparison purposes. Finally, we evaluate the results.

The final Section 7 presents the conclusion and the future work. We consider possible extensions of this work. In particular, we focus on how to get information about the usage of software systems.

[7]http://sourceforge.net/projects/ibatisjpetstore/
[8]http://www.eclipse.org/MoDisco, accessed 2010-09-27

# 2 Related Work

## 2.1 Software Aging

In [18], the author talks about the aging of software. He remarks that software gets older, in spite of software is a "mathematical product". He does not mean that software looses its mathematical correctness, he rather notes that legacy software systems handicap their owners over time. He compares the aging of software with the aging of humans and comes to the result, that aging is unavoidable, but it can be retarded. Even if we design software to meet new needs, we have to predict the future. But, this is only partly possible. Hence, we cannot stop aging. It does not only concern poor software design as Parnas states. He thinks, even “successful” systems get older.
He differentiates two causes of software aging. The first one is to miss reaction on changing requirements because of new business needs. Users get more and more sophisticated. Decades ago, people waited several days until computation finished for example, but today, http-requests have to be responded within milliseconds. So programs, which were common years ago, must be revised to keep relevant. Due to these changes, we get the second cause. Usually, software is designed by using simple concepts like design patterns. The original designer or programmer knows well these concepts. Extensions to existing software systems are often made by a third party. Consequently, knowledge about the design vanishes and changes will be inconsistent. If this is done many times, nobody knows about the current design and this may lead to more bugs and a longer development time.
Parnas mentions three issues which can slow down software aging. At design time, it is important to apply software engineering principles like “information hiding”, “abstraction”, or “separation of concerns”. If software is well-modularized, changes will only affect small pieces of the system and the original design can be kept. But it is not sufficiant to use an object-oriented language for meeting the principles. Such languages can help to

implement these rules, but the programmer is still responsible for the usage. The second issue is about documentation. He is of the opinion that documentation is essential for later changes. Although programmers often claim that their code is self-explanatory, it is mandatory to write a detailed documentation. Unfortunately, this is often neglected, since there is not a direct advantage. Finally, he recommends design reviews. At best, the reviewer is a person who is responsible for the long-term future of the software.
These issues only help, if the project is started on a green field. But there are many legacy systems, what can be done with them? First, the author suggests stopping the decline. The issues just mentioned can be applied. Each change must be checked against the origin design, must suffice the software engineering priniciples, and every change must be documented. The next step is to improve the existing documentation. This is not the most favorite work for programmers, but the enhancement of the documentation will upgrade next changes and therefore, the product owner should insist on a complete and traceable documentation. The last step is modularization, code snippets which belong together must be wrapped to modules.

## 2.2 Reengineering versus Reverse Engineering

[8] gives an overview about the used terminology. It is important to differentiate between reverse engineering and reengineering.
The authors define reverse engineering as a “process of analyzing a subject system to identify the system’s components and their interrelationships and create representations of the system in another form or at a higher level of abstraction“. Reverse engineering can be performed at any stage of the software lifecycle, while it is often applied on established systems. The process does neither include any changes to the software system, nor copying the base software system to create a new one. Reverse engineering will be done in order to improve comprehension of the software system and to make it easier to

change the system.
Reengineering, on the other hand, is ”the examination and alteration of a subject system to reconstitute it in a new form“. Hence, reverse engineering is rather the first part of reengineering, followed by forward engineering or restructuring.
The authors define forward engineering as ”the traditional process moving from high-level abstractions to physical implementations of the system."
Restructuring means the “transformation from one representation form to another at the same relative abstraction level”. The most common restructurings are code-to-code transformations to improve the structure of the code. Therefore, it is mostly not necessary to know about the design of the subject system. For example, to transform a set of “if” statements into a single “case” statement can be done without knowing about the purpose of the program. Usually, restructuring is not done because of new requirements but rather to improve the internal quality. So in general, restructuring does not change the behavior of the software system.

The object-oriented variant of restructuring is called refactoring. [17] gives an overview of refactoring activities. First, it is important to decide about the level of abstraction, which shall be refactored (the code directly, or design artifacts). Refactorings can be classified by the quality attributes they effect. Since refactoring does not alter the behavior there is an activity needed which ensures this. After that, the refactorings can be applied. Once this is done, the success must be evaluated. Finally, the consitency between all software artifacts must be maintained.

## 2.3 Software System Evolution

In [20], the authors state that system evolution of software systems can be classified into three sections. The first section is called maintenance. During this step only small changes will be made to the system, for example bug corrections or small functional enhancements. The second section is called modernization. This step contains major changes but still preserves a considerable percentage of the software system. If the system needs more invasive changes than can be done during maintenance, for example restructuring, then the system will be modernized. Finally, the third step which is called replacement deals with the rollout of a new system. This step must be used very carefully because it holds serious risks. Replacement needs many resources for developing the new software system. In contrast to established software systems, which are often well tested, it is not guarenteed, that the new system runs just as reliable.

[19] suggests a smooth migration of legacy systems. The authors explain a couple of reasons for this kind of migration. One reason is that it will not be possible for business companies to go out of business for a long time just because of launching a new system. The smooth migration supports an incremental replacement of business logic and client software. In [11], the Dublo pattern is described and the authors report on their experience with the smooth migration process.

In [14], the authors deals with the question of how to ensure, that the correct attributes are measured. Therefore, they propose an evaluation framework. Finally, they conclude with a detailed analysis of the attributes that lead to more useful data.

## 2.4 Software Metrics

The IEEE standard for a Software Quality Metrics Methodology [1] shows an approach to apply software quality metrics to software systems. It proposes the following five steps:

1. Establish software quality requirements
2. Identify software quality metrics

3. Gain commitment to the metrics set
4. Analyze the software metrics results
5. Validate the software quality metrics

It seems to be a very useful process which can also be applied with other software metrics.

Another possibility to establish a software quality model is stated in [7]. The authors give a short overview about the goal question metric (GQM) process and explain, that a GQM model is structured hierarchically. They propose a top-down approach to find appropriate metrics. First, goals must be defined. A goal contains information about the purpose of the measure, the object, which shall be measured, the issue to be measured, and the viewpoint from which the measure is taken [7]. Then, questions have to be found, which refine the goals, e.g. resource consumption optimization. Finally, metrics will be taken, which can answer the questions. Thereby, one metric can contribute to answer different questions.

# 3 OMG's Architecture-Driven Modernization Initiative

The Object Management Group (OMG) founded an Architecture-Driven Modernization (ADM) Task Force which should develop a set of modernization standards. The goal was, for example, to ease the understanding and evolving of existing software assets as well as consolidating best practices leading to successful modernization. It was founded in June 2003 and five months later, the first standard was recommended.

The OMG mentioned in their first Legacy Transformation Whitepaper different reasons, why it is important to understand and evolve legacy systems. Some of this reasons are software improvement, modifications, interoperability, refactoring, restructuring, reuse, porting, migration, translation, integration, and service-oriented architecture deployment. The union of these activities can be called Architecture-Driven Modernization.

In order to meet these needs, the task force aims to develop seven standard packages, each package serves a special task in the modernization process:

- Knowledge Discovery Meta-Model (KDM)
- Abstract Syntax Tree Meta-Model (ASTM)
- Pattern recognition
- Software Metrics Meta-Model (SMM)
- Visualization
- Refactoring
- Transformation

**Knowledge Discovery Meta-Model** represents entire software systems. This means, it does not represent only code but also structural and behavior elements. It is the base meta-model and other ADM standards depend on KDM.

**Abstract Syntax Tree Meta-Model** builds upon KDM. It aims at representing a software system below the procedural level.

**Pattern Recognition** examines structural meta data to find out patterns and anti-patterns within existing systems. These information can be used to determine refactoring requirements and opportunities.

**Software Metrics Meta-Model** aims to apply metrics to KDM instances. The metrics relate to different aspects of existing software systems such as functional, technical, and architectural issues.

**Visualization** focuses on ways to depict application meta-data.

**Refactoring** tries to define how to use KDM to refactor applications. It includes information about structuring, modularizing, and other ways to improve existing software systems.

**Transformation** defines mappings between KDM, ASTM, and a target architecture. So it is the bridge between existing software systems and target architectures.

Unfortunately, the Architecture-Driven Modernization standards are in an early state of development. Only the KDM is available in a stable version (V1.1). The SMM exists as a beta version (1.0) which is very unstable. We will consider this in detail in the next section. The ASTM also exists as a beta version (1.0). For all other standards only drafts exist, which are only available for OMG members.

In the following, we cover only the KDM and the SMM more in detail, since these meta-models are important for this thesis.

## 3.1 Knowledge Discovery Meta-Model (KDM)

The Knowledge Discovery Meta-Model specifies a set of common concepts required for understanding existing software systems. It is intended to represent information related

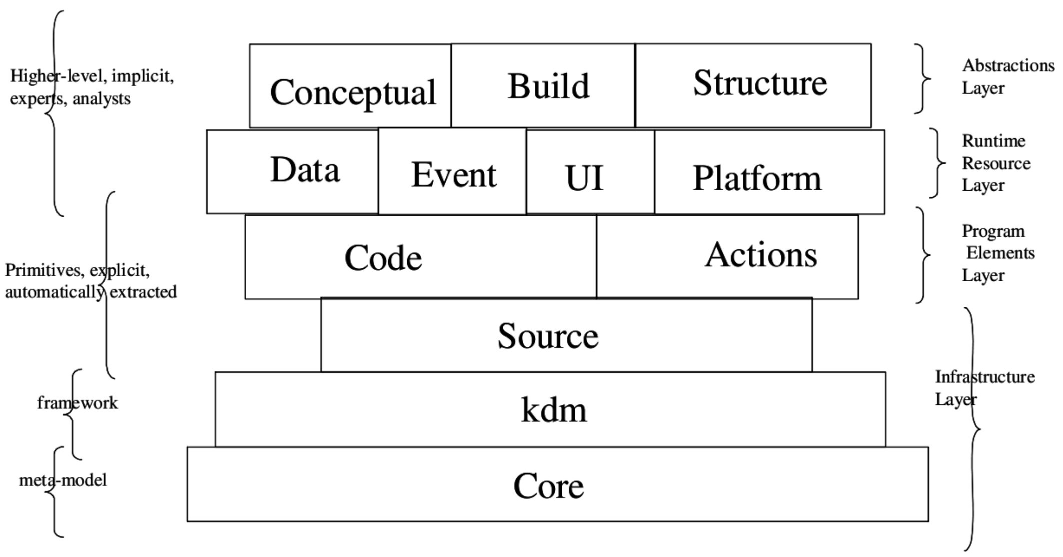


Figure 1: KDM package structure.

to existing software assets, their relations, and operational environments. It is possible to represent physical assets (e.g. source files) and logical assets (e.g. classes) at various levels of abstraction. Another goal is the exchange of models among a wide range of development tools. The KDM is a MOF (defined in [3]) model. So the KDM has got an XMI representation which can be exchanged.

Figure 1 (taken from [2], p. 11) shows how the Knowledge Discovery Meta-Model is designed. KDM consists of several packages grouped by layers. A layer depends on its underlying layers which means especially that all classes from packages above the core package inherit directly or indirectly from classes of the core package.
Now, we will describe the layers and their packages. We will only explain concepts which are important for this thesis. We do not want to create new kdm instances, rather we want to utilize existing ones. In this case, it will not be necessary to know about for example audit information represented by the KDM unit class like: "When was the model

generated or who created this element?"
The first layer is called Infrastructure Layer. Since it is possible to model different aspects of knowledge about existing software systems, the KDM is a large and complex specification. In order to handle this complexity, base concepts which are valid for all packages were concentrated in this layer. The layer consists of three packages, which are called core, kdm, and source.
Pursuant to the specification, from the view point of meta-modelling, the KDM follows the entity-relationship (ER) representation. An ER-model has got two elements, entities and relationships. Generally, an entity is a distinct identified thing with significance. Relating to software systems, an `KDMEntity` is an abstraction of an element of a software system. A relationship can be generally considered as an association between entities, which describes their interaction, their linkage, or their mutual dependence. So, a `KDMRelationship` represents the correlation between elements of software systems. So both classes are members of the core package.

The second package of this layer is the kdm package. This package contains elements for constructing KDM repesentations of existing software systems. It describes some elements which appear in every KDM instance. Every KDM instance is organized into segments which contain other segments or models.

There are nine different model types, which are all structured in the same manner. Figure 2 shows the structure with the help of the code package. The model is defined in the package of the same name, for example the `CodeModel` is defined in the code package. The model class is a concrete subclass of `KDMModel`. In the package an abstract parent for all entities related to this model is also defined. In our example, it is called `AbstractCodeElement`. The model class owns zero or more abstract elements. There are some more similarities, which we will not consider. Both, `KDMModel` and `Segment`, are specialized from `KDMFramework`. A segment contains a set of logically related KDM

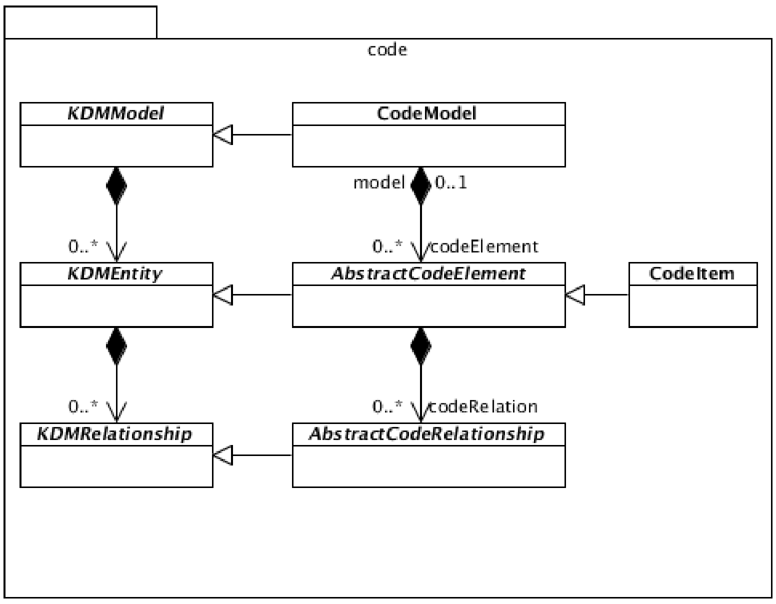


Figure 2: The general structure of KDM packages illustrated by the example of the code package.

models, which collectively provide a useful view of an existing software system.

The last package of the infrastructure layer is called source package. It contains elements which describe the physical artifacts of existing software systems. It also features the possibility to map KDM element instances to regions of source code. This is modeled by the class `SourceRef` which contains `SourceRegion` elements which further belong to a source file.

Now, we will introduce the program elements layer, which is very important for architecture metrics. This layer is divided into two packages called code package and action package. The code package provides a structural view on the existing software system, while the action package gives a behavioral view.

First, we want to consider the code package. It defines elements, which represent implementation level program elements and their associations. These program elements are for example data types, data items, classes, and so on. Generally every code package element belongs to a certain region of source code in one of the artifacts of the existing software system. But there are two exceptions. First, the `CodeModel` class is only a container for other code elements. Second, code elements, which represent certain abstractions of programming languages, such as data types, do also not belong to a region in the source code. This package is subdivided into five sections. The first section deals with code elements, which represent modules. Moduls are discrete and identifiable program units, for example a logical component of the software system. A concrete module is, for example, a package in a Java program. The second section is about callable computational objects. This is, for instance, a method or a procedure. The third section deals with elements which represent datatypes. In this case, datatypes are primitive data types (for example integer, char, boolean), or complex datatypes such as arrays or pointers. Finally a class or an interface (for example defined in Java) are also datatypes in terms of the Knowledge Discovery Meta-Model specification. The fourth section is about preprocessor directive elements. This is known from some programming languages, for example C++. The last section contains some code elements, which can not be assigned to any of the previous sections, for example the `Comment` class.

The second package in the program elements layer is called action package. This package extends the code package. So it does not provide a separate model class, rather it uses the `CodeModel` class from the code package. The main element in this package is called `AbstractActionElement`. Each action element is a unit of behavior. These elements can, similar to code elements, be mapped to regions in the source code of software systems. Action elements can contain other action elements. Such elements are called composite actions.

The most important concept in this package is the possibility to describe control flows between action elements. Therefore, it contains the `ControlFlow` class, which links two action elements. Another important concept in this package is the representation of exeptions.

The third layer is called Runtime Resource layer. It consists of the platform, the ui, the event, and the data package. The ui package provides elements, which contain information about user interfaces, for example their composition, their sequence of operation, and their relationships to the existing software systems. The main class is `UIResource`, which is an `AbstractUIElement` and contains other `AbtractUIElements`. This could be a screen, for example a webpage, which contains textfields or buttons. Since it is possible to model the sequence of operation, the flow between `AbstractUIElements` can also be modeled. The event package provides elements for the high-level behavior representation of software systems. The elements represent states, transistions, and events. An example for a state is a step of a protocol that involves a messaging resource. Transitions will be performed, after events have been executed in a certain state. The platform package deals with the representation of the runtime operating environments. The runtime platform is responsible for many different tasks. The platform could provide a middletier to access the database. The operating system is responsible to schedule the processes and must manage the access on the filesystem. The last package in this layer is the data package. It provides a view on the organization of the data in existing software systems. It uses parts of the code package related to the representation of simple datatypes. But elements of this package represent complex data repositories, such as record files, relational databases, or XML schemas.

The topmost layer is called abstraction layer. Since it is not important for this thesis, we only want to mention that this layer exists. It consists of three packages, which are called conceptual, build, and structure. Generally, this layer contains elements which represent domain-specific and application-specific abstractions. In the structure package, elements are defined, which model the architecture of the software system. In this context, a software system consists of subsystems, layers, or components. By using the build package, it is possible to give an engineering view of a particular system. This package also provides elements, which represent the artifacts which are generated by the build process. The last package of this layer is called conceptual package. It supports the creation of conceptual models during the analysis phase of knowledge discovery from existing software systems.

## 3.2 Software Metrics Meta-Model (SMM)

The main goal for creating the Software Metrics Meta-Model is to provide an extendable meta-model establishing a standard for the interchange of software-related measurements covering the entities modeled by MOF-conform meta-models. The SMM includes elements representing the concepts needed to express a wide range of software measures. The metrics convey technical, functional, and architectural issues which concern static as well as dynamic aspects. Furthermore, the meta-model contains a library of measures which does not constitute a standard, but rather conduces to show how to apply the SMM. This meta-model is also a MOF model itself. Hence, the meta-model has got an XMI representation which can be exchanged.

In the following we will explain the structure and the more important classes of the meta-model. Figure 3 (based on [4], p. 6) shows these classes. Every class in SMM with the exception of the measurands are specializations of the `SMM_Element` class.
The meta-model is intended to characterize special attributes of entities, e.g. complexity of software modules. Therefore, measures will be used to assess the traits and assign comparable values (numeric or symbolic) to the entities. A trait can be characterized by different measures which differ for example in the dimension. Each measure has got a

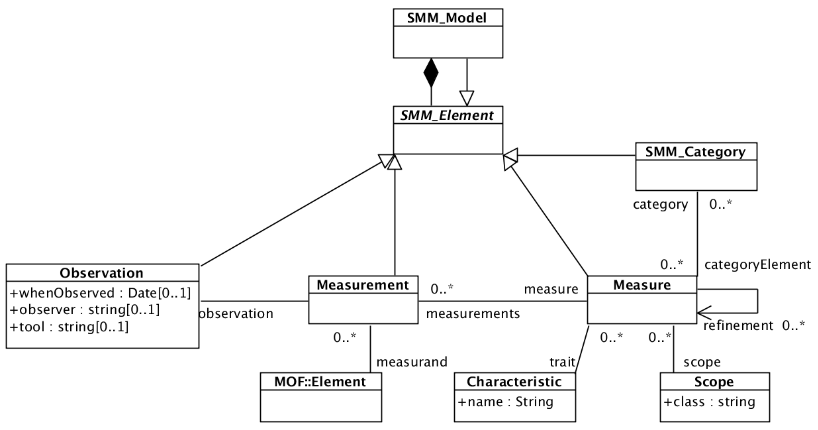


Figure 3: SMM core classes.

scope that comprises the set of entities to which the measure is applicable. Measures can be classified into different categories. For example "Lines of Code" can be related to the category "Length Measures". Furthermore, a measure can be refined by another measure which lastly means the refinement subsitutes the measure.
A measurement is a value assigned to an entity according to a measure. The measurement class represents the results of applying the associated measure to the associated measurand. Both `Measure` class and `Measurement` class are abstract and are extended by a couple of classes.
Every measurement is described by an observation which contains additional information, e.g. the date when the measurement was accomplished.
According to the meta-model, measurands can be any MOF object. This is a very important fact because the KDM is MOF-conform. So we can associate measurements to elements of the KDM by applying measures.

Figure 4 (based on [4], p. 10 and p. 16) shows how `Measure`s are structured. First, a measure can be a `DimensionalMeasure` or a `Ranking`. `Ranking`s represent range-

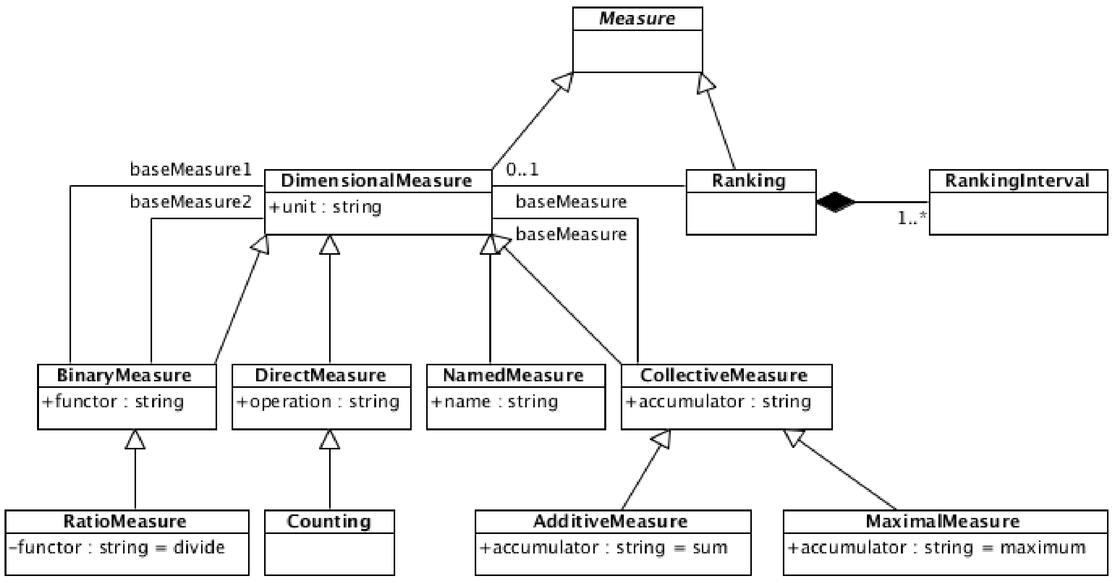


Figure 4: Measure classes in the SMM.

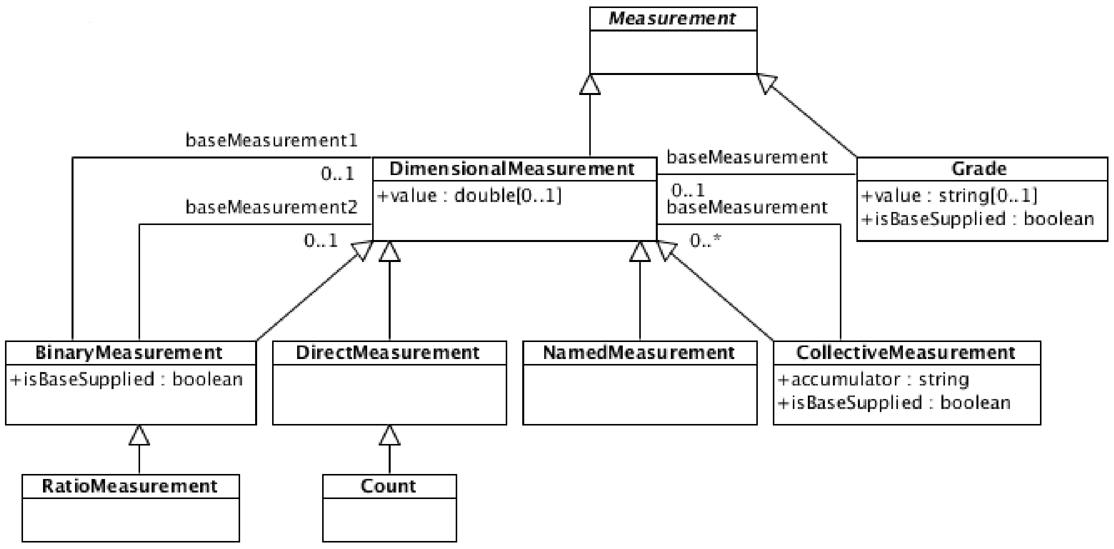


Figure 5: Measurement classes in the SMM.

based gradings such as low, medium, high. Therefore, a `Ranking` has at least one `RankingInterval`. So, a range of numerical values can be mapped to an interval. The intervals can overlap or can have gaps. This means, if a ranking results in an particular interval, it will not exclude, that the ranking could result in another interval. Usually, a ranking measure hase got a `baseMeasure`. But, it do not necessarily have a `baseMeasure`. This happens, if the measure represents a qualitative attribute, which does not need a quantitative evaluation. One example is the classification of the used programming language.

`DimensionalMeasure`s can be split into four groups: Members of the first group are `BinaryMeasures`, which have two `baseMeasures`. The `BinaryMeasure` applies the `functor` to the `baseMeasures`. The specialized class `RatioMeasure` takes the result of `baseMeasure1` and divides it by `baseMeasure2`. For example, let `baseMeasure1` be the result of counting all methods in a module and `baseMeasure2` the result of counting all classes in the module. If we apply the `RatioMeasure`, the result will be a methods per class measurement. The second group consists of `DirectMeasures`. These measures will be applied directly to measured entities. The given operation has to return a numeric value. If the `DirectMeasure` is a `Counting`, then the result hast to be 1 or 0, depending on whether the measured entity was recognized, or more precise whether the measured entity is an instance of the class defined in the scope of the `Counting` class. A common example for the `Counting` measure is to count methods in a module. If a method was found, the `Counting` class will return 1 and the sum of all `Counting` results is the method count. The third group contains `NamedMeasure`s, which do not need an operation as the measure can simply be specified by the given name. We will see the utilization in Section 7. The last group consists of `CollectiveMeasure`s. If a measured entity is an instance of the class defined in the scope, then the `baseMeasure` will be applied to this entity and to all contained entities. Then the n-ary `accumulator` will be applied to all base measurements and the result will be assigned to the measurement of this entity. We will

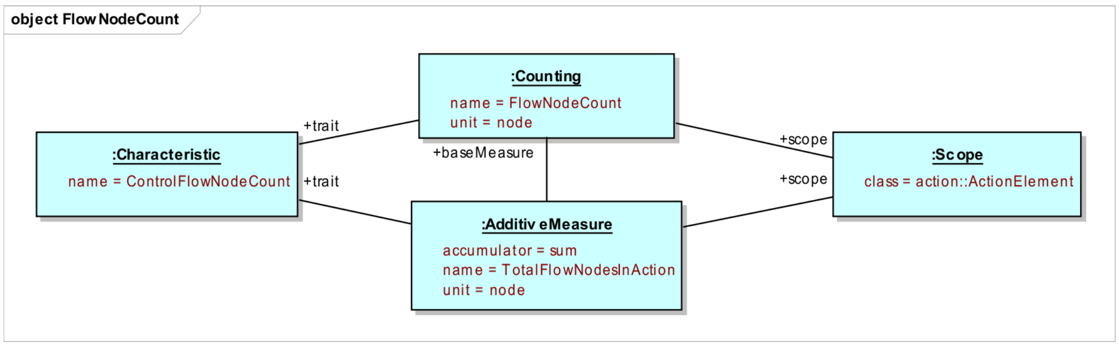


Figure 6: Control Flow Node count

give an example at the end of this Section.

Now, we examine measurements in greater detail. For every measure, there is defined an adequate measurement. Hence, the reader can see in Figure 5 the same hierarchy as for measures. In the specification it is explicitly defined, that the measure associated to a `BinaryMeasurement` must be a `BinaryMeasure`. This is analogically valid for all other measurements.
For all measurements, which possibly can have `baseMeasurements`, there is defined an attribute which is called `isBaseSupplied`. If we look at Figure 4 again, we see, that a `CollectiveMeasure` has got mandatorily a `baseMeasure`. `CollectiveMeasurements` do not have this restriction. Let us assume, that the class which is in the scope of the `CollectiveMeasure` does not contain any class which is in the scope of the `baseMeasure`, then, there will not exist a measurement for the `baseMeasure` and consequentially, there is no `baseMeasurement` for the `CollectiveMeasurement`.

Figure 6 shows an example and figures out how to model measures. It is part of the library which is defined in the specification ([4], p. 47). It defines a measure which counts the nodes of a control flow modeled in KDM. This measure is one necessary part of McCabe's cyclomatic complexity [16]. The classes `Counting` and `AdditiveMeasure` are specializations of the `Measure` class. Every measure has got a `Scope` which indicates on which elements the measure can be applied. In the example, the measure is related to the `ActionElement` class which is defined in the action package of the KDM. Finally, every measure has got a `Characteristic` which describes the purpose of the measure.

# 4 Contribution to the CloudMIG Platform

CloudMIG is an approach for the migration of existing software systems to the cloud and provides an accompanying identically named Eclipse Rich Client Platform application (RCP). Eclipse was originally only an Integrated Development Environment (IDE) which supported building Java programs. Later, all components of the IDE which are generally necessary for standalone clients were extracted and combined to the Rich Client Platform. Thus, it is the minmal set of plugins supplying a general infrastructure for desktop clients. RCP based applications are independent from the original IDE but they are still using the Eclipse UI and dynamic plugin-model. Generally, Eclipse-based applications can be extended by plugins to enlarge the functionality. A plugin can be seen as a component of the software. Since it is desirable to have loose coupling between components, Eclipse provides a generic mechanism of extensions and extension points. When a plugin allows to extend or customize some of its functions, then it provides an extension point, which declares a contract, usually a combination of XML[9] markup and Java interfaces or classes. If a plugin wants to connect to this extension point, it must define an extension and has to implement the contract [12]. We used this mechanism to contribute our plugins to the CloudMIG platform.

## 4.1 Architecture

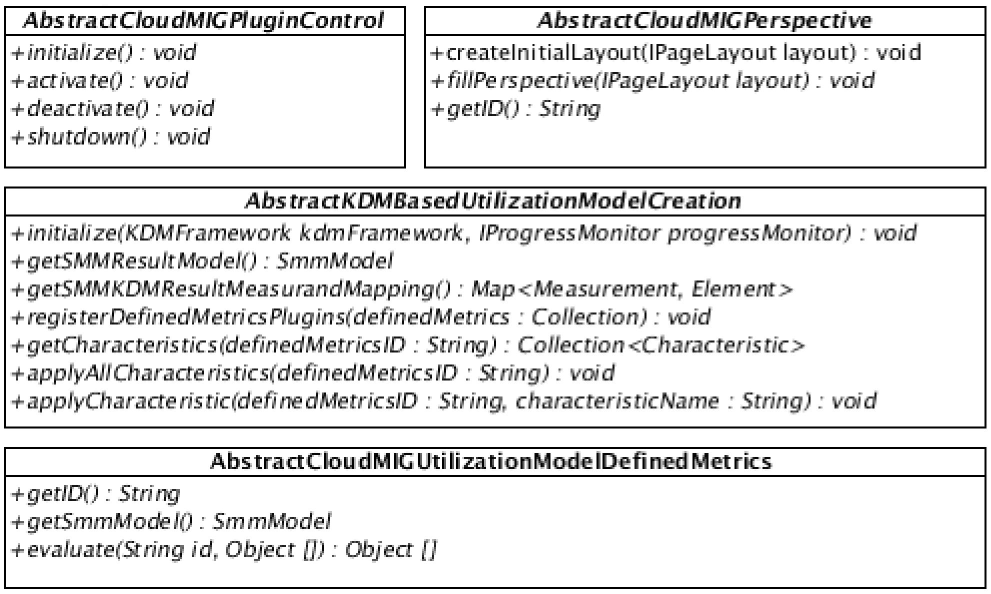
In order to apply metrics to given resources, such as KDM instances, log files or to import externally applied metrics, CloudMIG provides one important extension point, called *utilizationmodel*. It will be extended through all plugins which contribute to the application of software metrics. This includes plugins, which support the definition of metrics, further it includes plugins which apply metrics and it includes plugins which support this process. The extension point defines domain-specific elements and to every element belongs an interface or class must respectively can be realized or rather extended. Figure 7 shows the

[9]Extensible Markup Language

abstract classes which must be extended through our plugins. The first element is mandatory and is called *Control* and the related class is `AbstractCloudMIGPluginControl`. It manages the lifecycle of the plugin. Therefore, the class provides abstract methods for initializing, activating, deactivating, and shutting down the plugin. The next element must only be implemented by plugins which need access to the user interface. The element is called *Perspective* and relates to the class `AbstractCloudMIGPerspective`. The plugin can register views and editors with the method `fillPerspective(IPageLayout layout)`. Finally, there is again a mandatory choice between three elements. It indicates of which type the plugin is. The first element is *KDMBasedCreation*. Plugins which use this element, extend the class `AbstractKDMBasedUtilizationModelCreation`. They are responsible for the application of the metrics, thus the class has the methods (among others) `applyAllCharacteristics(String definedMetricsID)` and `applyCharacteristic(String definedMetricsID, String characteristicName)`. The second element is called *MetricsDefinition*. It is not necessary for this element to extend a class. Plugins which use this element, facilitate the definition of metrics. *DefinedMetrics* is the last element. If this element was chosen by a plugin, the class `AbstractCloudMIGUtilizationModelDefinedMetrics` was extended. Plugins which use this element contain defined metrics saved as a SMM instance. Furthermore, the plugins contain a helper class which expands the SMM model instance.

In Figure 8 the plugins we provide are shown and how they are linked. The notation of the diagram is according to UML[10] component diagrams. In this case the port corresponds to an Eclipse extension point which will be extended throug all plugins of the tool. We implemented one plugin called *Metrics Definition*, which allows the user to define SMM conform software metrics. Therefore, it contains an editor. This plugin creates new plugins which contain the defined models and a helper class. Finally, we provide a

[10]http://www.uml.org/, accessed 2010-09-27



```
AbstractCloudMIGPluginControl
+initialize() : void
+activate() : void
+deactivate() : void
+shutdown() : void

AbstractCloudMIGPerspective
+createInitialLayout(IPageLayout layout) : void
+fillPerspective(IPageLayout layout) : void
+getID() : String

AbstractKDMBasedUtilizationModelCreation
+initialize(KDMFramework kdmFramework, IProgressMonitor progressMonitor) : void
+getSMMResultModel() : SmmModel
+getSMMKDMResultMeasurandMapping() : Map<Measurement, Element>
+registerDefinedMetricsPlugins(definedMetrics : Collection) : void
+getCharacteristics(definedMetricsID : String) : Collection<Characteristic>
+applyAllCharacteristics(definedMetricsID : String) : void
+applyCharacteristic(definedMetricsID : String, characteristicName : String) : void

AbstractCloudMIGUtilizationModelDefinedMetrics
+getID() : String
+getSmmModel() : SmmModel
+evaluate(String id, Object []) : Object []
```

Figure 7: CloudMIG utilizationmodel extension point.

plugin, which applies defined metrics to given KDM instances. Its main component is the *Metrics Computation* component.

## 4.2 Metrics Definition Plugin

This plugin facilitates the definition of metrics. We use the terminology "full metric definition", if we talk about an Eclipse plugin which consists of a SMM instance and of a helper class. As we are in the Eclipse domain, the helper class is a POJO[11]. The helper class consists of methods which can be referenced in the SMM instance, for example as an operation to be executed. So, the plugin needs two editors, one for defining SMM instances and one for editing the Java class.

[11]Plain Old Java Object

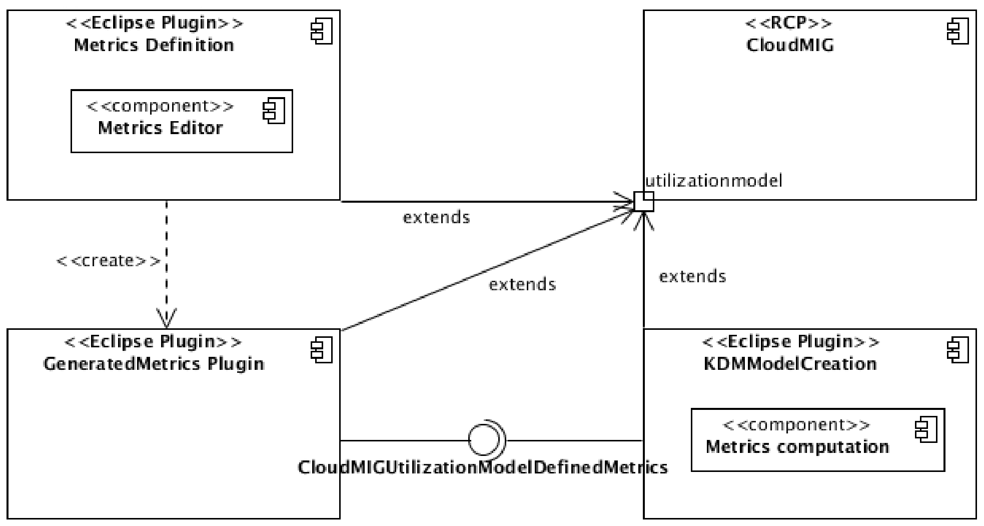


Figure 8: Architecture of the tool

Since Eclipse was originally a Java IDE, it comes with a Java editor out of the box which supports, among other things, syntax highlighting or code completion. Furthermore, Eclipse facilitates the programmatic generation of so called projects, especially Eclipse plugins. It simplifies the programmatic plugin generation process as we will see below. Fortunately, there exists an EMF-conform SMM implementation which is provided by MoDisco. As mentioned earlier, EMF provides tools for code generation which facilitate the usage of model instances. Hence, we can use it to generate a tree-based editor, which makes it very easy to define SMM models. It also provides a validator which checks the defined model against the specification. However, it does not cover the full specification, as the model provided by Modisco is not complete.

The plugin supports the creation of new metrics definitions as well as the editing of existing ones or the import of external SMM instances. We have to differentiate the last two ones, because, as mention above, a metrics definition is a complete plugin while the import refers to a single SMM instance. This plugin also supports the generation of a JAR[12] package which contains the defined metrics. These archives can be moved into the plugin folder of the CloudMIG Platform and then, they can be used for the metrics application.

## 4.3 KDM Model Creation

This plugin is responsible for the application of metrics. The main component of this plugin is the execution engine which we will describe in detail in Section 5.2. CloudMIG assigns all metrics definitions, which are located in the classpath of CloudMIG, to this plugin. The plugin itself provides a user interface which allows the user to select SMM characteristics which are defined in the metrics definition plugins.

Another configuration which must be done, concerns external libraries. A KDM instance usually contains a `CodeModel` object which is named *externals*. It contains information about external libraries which are used in the actual project. If a SMM instance is applied to this KDM instance, it will comprise this `CodeModel` object. This may contort the result and therefore, we provide the possibility to ignore the external data.

After configuring the computation, the execution engine will be invoked. It applies the defined metrics and creates a result model. This model is also a SMM instance which contains measures and their measurments. Furthermore, the execution engine creates a mapping between the measurements and the measured KDM elements as this relation is defined in the SMM specification but is not implemented in the meta-model provided by MoDisco.
Finally, CloudMIG can access this data through the defined extension of the plugin. So this data can be used for further computation.

[12]Java Archive

# 5 Process of applying generic SMM models

## 5.1 Flaws in the Specification

Since the Software Metrics Meta-Model is only available as a very early version, many flaws exist in the current version 1.0 Beta of the specification. We want to discuss the most significant ones and try to give advices how to cope with the problems.

#### Consistency

While reading the specification, many mistakes appear. It starts with simple “copy and paste” errors like on page 18: In the constraints part of the class `AdditiveMeasure`, there is the context `MaximalMeasure`, which was simply copied from the `MaximalMeasure` class. It proceeds with wrong headlines of the sections. In our opinion, section 10 deals with `DimensionalMeasures`, whereat the mentioned `CollectiveMeasure` is only a subsection. Furthermore, examples are wrong. In Figure 7 int the specification, there is an association between `Characteristic` and `Scope`. Pursuant to Figure 3 in the document, there is not a direct association between these classes, rather they are linked by the `Measure` class. Finally, there are classes, which are not defined. Usually, every class which is mentioned in a class diagram is defined in a text. In Figure 8 in the specification, the class `AggregatedMeasure` is mentioned, but there is not a definition about this class. It is not clear, if the class diagram contains obsolete elements, or if the definition was forgotten. We decided not to implement this class, as we can only guess about the functionality.

#### Overhead

The class `CollectiveMeasure` is subclassed by `AdditiveMeasure` and MaximalMeasure. They only differ in their accumulator. Hence, we suggest to create an Enumeration

Accumulator, which contains the different accumulators as elements. Then, only the class CollectiveMeasure with the attribute accumulator is needed. This is not necessarily a flaw, rather an improvement in order to implement the execution engine. Each additional measure leads to a more complex execution engine, because each measure needs its own workflow. Moreover, this meets the next problem:

### Operations

This problem relates to all measures, which allow operations, functors, accumulators, or aggregators. The library suggests to define a `DirectMeasure` with the following operation: `operation=endLine-startLine+1` in which `startLine` and `endLine` relates to the KDM element `SourceRegion`. It could seem that nothing more must be done, but it is not sufficient. This can only be an id for a operation, that the user has to define seperately. Alternatively, this can be an OCL[13] statement. Unfortunately, this is not clearly specified.

### Insufficient Description

The entry point for the application of measures is the defined characteristic. A characteristic has got different measures. As we have seen, after application of the measures, every measure relates to a measurement result. But which measurement result represents the result for the characteristic. This is unfortunately defined nowhere. After we analyzed the library of the specification, we assume, that the measure which is not a base measure for any other measure, belongs to the measurement which contains the result for the characteristic. Furthermore, the modeler must be assure, that there is exactly one measure which has this property. We included these assumptions in the execution engine.

The last problem we want to mention, is the description of the measurement process

[13]www.omg.org/spec/OCL/2.0/, accessed 2010-09-27

for collective measures. The specification explains, that the base measure has to be applied to each contained element. It is unclear, what is meant by "contained element". It could be the directly contained elements or also the recursively contained elements. We implemented the latter, as it is used in the library that way.

## 5.2 The Execution Engine

In this Section, we want to explain the operating principle of the execution engine's measurement process. This process will be executed once for each characteristic which was selected in the user interface. Generally, this process can be applied to all MOF-conform model instances. In order to describe the process more understandable, we assume that the model which will be measured is a KDM instance. The process can be divided into three phases, the "initialization phase", the "recursive measure application phase", and the "assignment phase".
During the first phase, the engine has to discover different elements. As mentioned above, there exists a measure which is not the `baseMeasure` for any other measure. In the following, we will call this measure "main measure", because this measure is the entry point to get the measurement results for a characteristic. In order to find the measure, the execution engine has to search all measures which are associated to the characteristic. During the initialization phase, the second step is to find all elements to which the main measure will be applied. To each measure exactly one scope element is assigned. The scope element has got an attribute which is called `class`. It defines the type of the elements to which the measure will be applied. So, the execution engine has to get the class and after that it searches the complete KDM instance to discover all elements of this type.

After that is done, the second phase starts and the main measure will be applied to all discovered elements. The same workflow must be processed for every measure. Figure 9 shows this workflow which consists of two parts. First, an adequate measure must be created. As we have seen in Section 3.2, to each measure belongs exactly one measurement type. So this is meant by “adequate”. If the measure is an instance of `Ranking`, then the adequate measurement will be an instance of `Grade`, for example. The second part deals with the application of the measure. Here, it must be distinguished beetween the type of the measure, because a `Counting` needs another treatment than a `CollectiveMeasure`. We will see the different treatments later in this Section.

Prior, we deal with the third phase. After the measure was applied, the result of the application must be assigned to the measurement. Furthermore, it is necessary to create and assign an observation object to the measurement. An observation contains some meta data such as date of the execution or the person who invoked the measurement process. Finally, the mapping between the measurement and the measurand must be realized.

In the following, we will describe the treatments of the different measures. Therefore, we assume that we apply a measure $M$ to an element $E$. It is important that the order of the type checking will be the same as in Figure 9.

### Counting

First, it must be checked, if $M$ is an instance of `Counting`. In this case the defined `operation` will be applied to the given element $E$. The `operation` returns a result which can be assigned to the related `Count` element. If no `operation` was defined, then the result is always 1.

### DirectMeasure

If $M$ is an instance of `DirectMeasure`, the application is the same as for `Counting` with the exception that the result will be assigned to a `DirectMeasurement` and the result can be any numerical value.

### RatioMeasure

If $M$ is an instance of `RatioMeasure`, first the `baseMeasure1` will be applied to the given element $E$. This means that the measurement process we described before must be executed recursively. Then, the `baseMeasure2` will be applied analogically. Finally, the result of `baseMeasure1` will be divide by the result of `baseMeasure2`.

### BinaryMeasure

If $M$ is a `BinaryMeasure`, the same workflow as for `RatioMeasure` is valid, with the exception that the last step is more general. It is only specified that the defined `functor` will be applied to the results.

### CollectiveMeasure

If $M$ is an instance of `CollectiveMeasure`, then the `baseMeasure` must be applied recursively to all contained elements of $E$. What is meant by "contained elements" was described before. After the execution engine finished the recursive application, the results must be aggregated. This will be done by using the defined `accumulator`. For example, if the `accumulator` has the value `sum`, then all computed results will be added.

### RescaledMeasure

In the case that $M$ is an instance of `RescaledMeasure`, first the `baseMeasure` must be applied recursively to the element $E$. After that, the `formula` will be applied to the computed result.

### Ranking

Finally, if $M$ is an instance of `Ranking`, first the `baseMeasure`, if existent, will be recursivly applied to the element $E$. After that, the execution engine will look up an

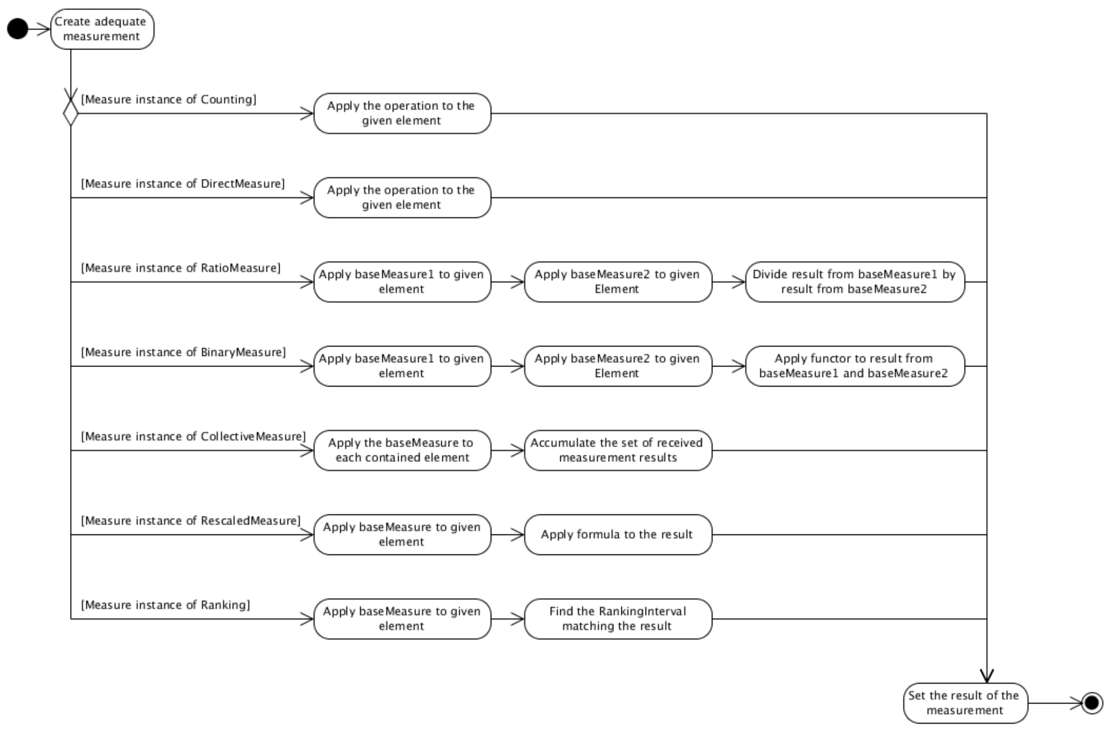

Create adequate measurement
[Measure instance of Counting]
Apply the operation to the given element
[Measure instance of DirectMeasure]
Apply the operation to the given element
[Measure instance of RatioMeasure]
Apply baseMeasure1 to given element
Apply baseMeasure2 to given Element
Divide result from baseMeasure1 by result from baseMeasure2
[Measure instance of BinaryMeasure]
Apply baseMeasure1 to given element
Apply baseMeasure2 to given Element
Apply functor to result from baseMeasure1 and baseMeasure2
[Measure instance of CollectiveMeasure]
Apply the baseMeasure to each contained element
Accumulate the set of received measurement results
[Measure instance of RescaledMeasure]
Apply baseMeasure to given element
Apply formula to the result
[Measure instance of Ranking]
Apply baseMeasure to given element
Find the RankingInterval matching the result
Set the result of the measurement

Figure 9: The execution engine's measurement process

`RankingInterval` matching to the result. Then, the value of the interval will be assigned to the measurement.

This execution engine can be extended in order to support new measures. Only the workflow must be integrated in the execution engine and the creation of an adequate measurement must be customized. It is required that the new measure will be integrated at the right place. Generally, the most specialized measures must be checked first and the most general measures must be checked at the end. Hence, Figure 9 shows the current order of the execution engine. First it will be checked if the measure is an instance of `Counting` which is a specialized measure. At the end, one of the most general measures will be checked.

# 6 Evaluation

## 6.1 Methodology

This Section describes the evaluation of the developed plugins. We decided to compare the results computed by our tool to the results computed by an established tool. The reference tool is an Eclipse plugin called *Metrics*[14]. It provides a set of metrics, which can not be changed or extended. We took three different metrics which can be computed. The first one is called *ClassCount*, the second one is *Lines of Code* and finally, we apply the metric *Depth of Inheritance Tree*. The three metrics will be applied to three different software systems. The first one is a simple program without any practical use. The second software system will be *NanoHTTPD* which is a simple HTTP server also written in Java. It consists of only one Java file which contains one regular class and two nested classes. Finally, we apply the metrics to the *iBATIS JPetStore* which is a Java web application. It is an online shop where the user can buy different pets. This is a more complex software system compared to the other examples. We utilized MoDisco in order to get KDM instances of the software systems. Unfortunately, these instances did not contain any `SourceRef`s which made it impossible to apply the lines of code measure. Hence, we used a further tool which creates also KDM representations of software systems. This tool is called KADis[15].

## 6.2 Selected Metrics

### ClassCount

This measure simply counts all classes in the project. It is not a very complex metric but it can provide information about the size of the software system.

[14]http://metrics.sourceforge.net/
[15]http://sourceforge.net/projects/kadis/

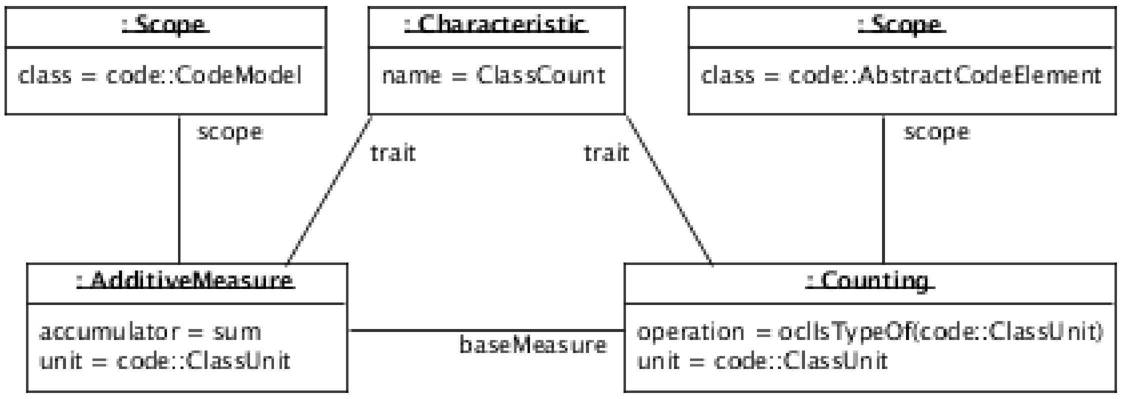


Figure 10: Class Count Measure.

Figure 10 shows the representation of this measure in SMM. In KDM, a class is represented by a `ClassUnit` which is a subclass of `AbstractCodeElement`. These elements are contained directly or indirectly in a `CodeModel` element. “Directly” means that the `ClassUnit` element is a child of the `CodeModel` element, whereas “indirectly” means that the element is not a child but rather a descendant of the `CodeModel`.

First, the execution engine has to find all `CodeModel` elements in the KDM instance. Then, the `Counting` measure will be applied to all directly or indirectly contained elements of the discovered code models which are of the type `AbstractCodeElement`. If such an element is found, the defined `operation` will be applied to this element. If the element is of type `ClassUnit`, the result will be 1 otherwise it will be 0. Finally all results related to the `CodeModel` element will be accumulated.

### Lines of Code

The second measure accumulates the lines of code of the software system. The result of this measure strongly depends on different parameters, such as the used programming

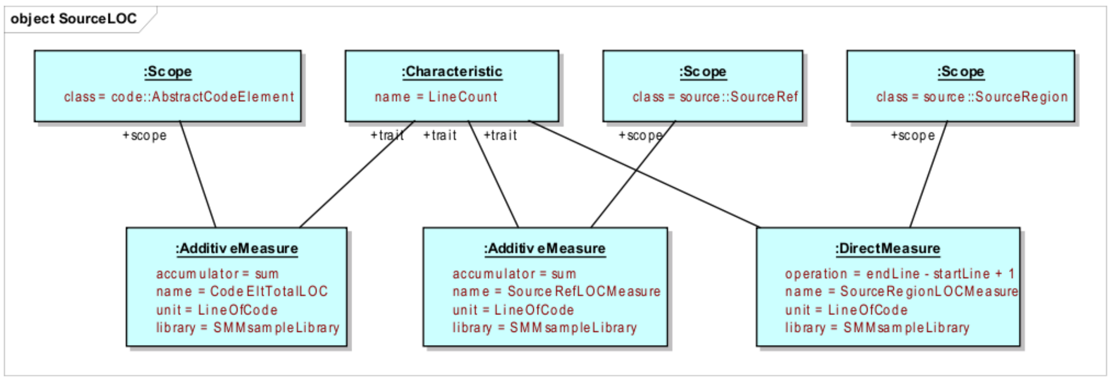


Figure 11: Lines of Code Measure.

language and the habit of the programmer. Hence, the result must be carefully interpreted.

Figure 11 (taken from [4], p. 42) shows the SMM representation of this measure. Every `AbstractCodeElement` represents a software artifact which can be mapped to a region in the source code. Therefore, each element has got a child which is of type `SourceRef`. In such an element, there is a `SourceRegion` element defined, which repesents among other things the startline and the endline of the code snippet.

Hence, the execution engine has to find all `AbstractCodeElement` elements in the KDM instance. Afterwards, the execution engine applies the `baseMeasure` to all children, if they are of the type `SourceRef`. Then the `Counting` measure will be applied to all children of the `SourceRef` element which are of type `SourceRegion`. If such an element is found, the operation will be applied to this element and the result is the number of lines related to the code element.

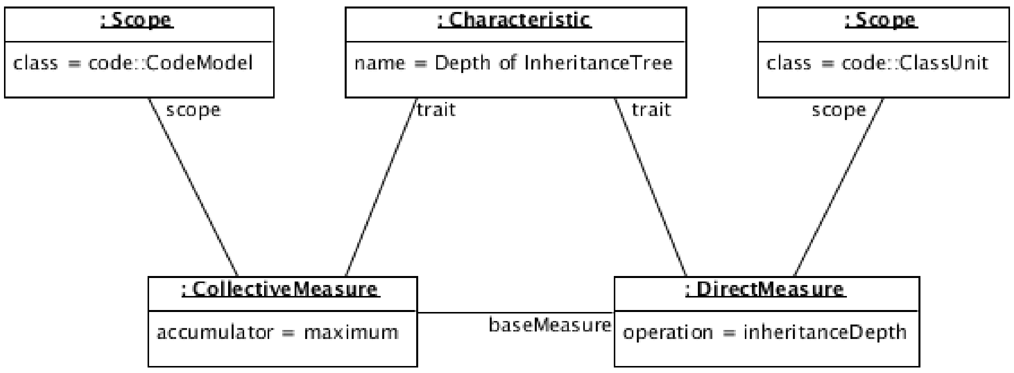


Figure 12: Depth of Inheritance Tree

### Depth of Inheritance Tree

The third measure computes the depth of the inheritance tree for a software system. This measure provides information about the level of abstraction.

Figure 12 shows the related SMM instance. It is similar to the classcount measure. First, the execution engine has to find all code model classes and has to apply the `baseMeasure` to all directly and indirectly contained elements.

Hence, the execution engine has to find all code models to which the main measure can be applied. Then, it has to discover all `ClassUnit`s and has to apply the defined `operation` to this element. The result of the `operation` will be the depth of inheritance for this class. Finally, the maximum for all classes will be selected.

## 6.3 Analysis

After the definition of the measures, we apply them to the software systems. Figure 13 shows the results of the application. Most of the measurement results match but

| Project | CC | LOC | DIT |
|---|---|---|---|
| Simple Java Project | 6* | 81* | 2* |
| | 6** | 25** | 2** |
| NanoHTTPD | 3* | - | 1* |
| | 3** | - | 1** |
| iBATIS JPetStore | 36* | - | 5* |
| | 36** | - | 5** |

* Computed by *Metrics*

** Computed by the developed tool

Figure 13: Measurement results of the Evaluation

we see also a difference in the computation of lines of code. After a closer look at the provided source KDM instance the problem was discovered. Unfortunately, the instance does not represent the software system completely. Many action elements do not contain `SourceRef`s. For this reason many code snippets have not been included. For example the `import` instructions for a class were not mapped to the source code. This leads to the mismatch between the measurement results. Hence, we did not apply the measure to the other projects as the results would also be wrong.

So, we can conclude that the tool will work as expected, assumed that *Metrics* provides correct results. We have seen that 6 of 7 tests supplies the expected results. Only one test fails because of the underlying KDM instance was not sufficient. This means, if we have a suitable KDM instance, it seems that the results will be correct.
However, we have to remark, that we only could test a restricted number of measures and projects. Hence, the evaluation has a limited validity.

# 7 Conclusion and Future Work

As we have seen, software aging is an important issue, but it is possible to slow it down. This is closely connected with the topic software reengineering which is a combination of reverse engineering followed by forward engineering or restructuring. The OMG tries to support this process by introducing its Architecture-Driven Modernization approach. ADM conduces to facilitate the whole modernization process and hence, it contains of a set of standards, which are helpful. We described the Knowledge Discovery Meta-Model which aims to represent different views of a software system. Furthermore, we covered the Software Metrics Meta-Model which can be used to represent measures and their measurement results. Unfortunately, the SMM specification is only available as an early version and for this reason, it is partially inconsistent or unclear. Nevertheless, it can be used to apply metrics programmatically. We introduced our execution engine and we described how to handle the existing flaws. It is necessary to define some restrictions in order to use SMM, because it is not possible to translate every generally defined attribute into adequate software snippets. We remarked, that every measure, defined in the SMM, must be specifically handled which means that the execution engine consists of different routines for the measures. We used the SMM to represent metrics and their results after applying these metrics. Moreover, we contributed Eclipse plugins to the CloudMIG platform. These plugins support the complete process of metrics application. This starts with the definition of metrics, proceeds with the choice of appropriate metrics, and ends with the application of the chosen metrics. Finally, we evaluated the execution engine by dint of the established Eclipse plugin *Metrics* and several open source systems. If the used KDM instance supplies the needed information then our tool returns the same results like the *Metrics* plugin with different grades of complexity. However, in contrast to *Metrics*, our tool is easily extensible by providing a universal execution engine which can handle arbitrary SMM-conform metrics.

Currenty, it is only possible to apply metrics to KDM instances. This implies that either the KDM instance exists or the source code is available and the KDM instance can be computed. If none of the two is possible, there are maybe other alternatives to get measured information about the software system. There are many other tools which compute metrics and maybe such results exist for the software system. These results could be integrated in a SMM instance very easily. As we have seen in Section 3.2, `NamedMeasure` can represent externally computed measurement results. In order to use this functionality in CloudMIG, we suggest to create a new Eclipse plugin which also extends the extension point *utilizationmodel* by using the attribute *ExtCompMetricsBasedCreation*. This plugin must supply a mechanism to parse externally computed metrics results. Furthermore, it has to create a new SMM instance, add a set of `NamedMeasure`s which represent the names of the metrics and add a set of `NamedMeasurement`s which represent the results of the metrics. An important question in this context is, if it possible to provide a generic solution for a parser.

Another question is how to incorporate external information about the usage of a software system. We have seen, that SMM can be used to model a wide range of metrics, this includes usage metrics. We can use our developed tool to model usage metrics (see the example in Figure 15). Now, it is necessary to apply these metrics to existing sources. An exemplary source of usage data are log files. They often contain information about method calls or webpage calls. We suggest the creation of a meta-model which represents log files. If this meta-model would be MOF conform, it had been possible to apply SMM instances to this model. Unfortunately, log files often do not suffice a standard, which makes it difficult do find a generic solution. Hence, it will be a challenge to find an appropriate meta-model. If such a model was found, the execution engine, presented in Section 5.2, could be applied to the model.

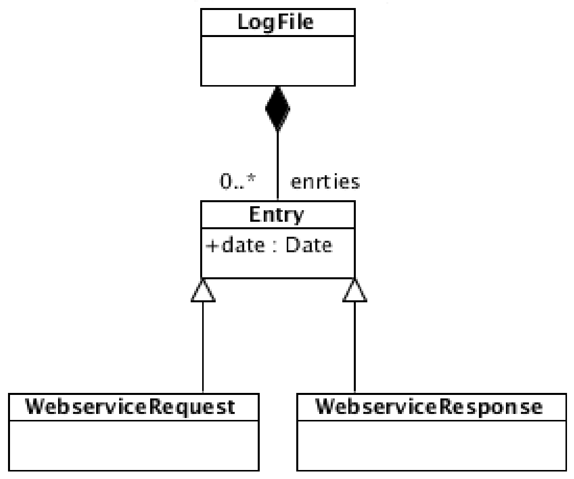


Figure 14: Easiest Log File Meta-Model.

Figure 14 shows a very easy meta-model of a log file. In this case, a log file consists of many entries. An entry itself, represents different kinds of log file entries. Let us assume, that webservice requests will be logged, then every entry which represents a webservice request is of type `WebserviceRequest`.
Now, we can define a metric called *WebserviceRequestCount* which counts all webservice requests logged in a file. Figure 15 shows the SMM instance. It is anlogically defined to the *ClassCount* measure in the Section 6. Every occurence, of a webservice request will increase the number of total webservice requests.

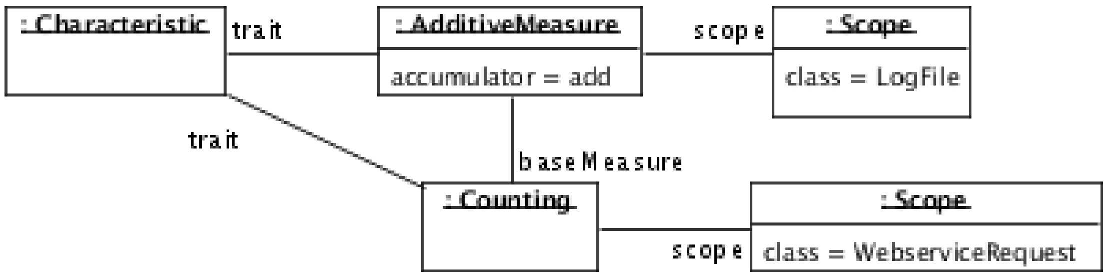


Figure 15: Webservice Request Count.

# A Attachments

We have attached a CD which contains four folders. In the folder “analyzed projects” there are the three projects which we have used for the evaluation. The folder “Defined Metrics” contains the metrics we used for the evaluation. The folder “Eclipse plugins” contains the two plugins we contributed to the CloudMIG platform. Finally the last folder which is called “plugins” contains additional plugins which are used by our own plugins.

## References

[1] IEEE Standard for a Software Quality Metrics Methodology, December 1992.

[2] Architecture-Driven Modernization (ADM): Knowledge Discovery Meta-Model (KDM) OMG Adopted Specification, January 2009. URL `http://www.omg.org/spec/KDM/1.1/`. [Online; accessed 2010-09-27].

[3] Meta Object Facility (MOF) Core Specification, January 2006. URL `http://www.omg.org/spec/MOF/2.0/`. [Online; accessed 2010-09-02].

[4] Architecture-Driven Modernization (ADM): Software Metrics Meta-Model (SMM), March 2009. URL `http://www.omg.org/spec/SMM/1.0/Beta1/`. [Online; accessed 2010-09-27].

[5] IEEE Standard for a Software Quality Metrics Methodology. *IEEE Std 1061-1998*, page i, dec. 1998. doi: 10.1109/IEEESTD.1998.243394.

[6] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, and M. Zaharia. Above the clouds: A berkeley view of cloud computing. Technical report, 2009.

[7] V. R. Basili, G. Caldiera, and H. D. Rombach. The goal question metric approach. In *Encyclopedia of Software Engineering*. Wiley, 1994.

[8] E. Chikofsky and I. Cross, J.H. Reverse engineering and design recovery: a taxonomy. *Software, IEEE*, 7(1):13 –17, jan. 1990. ISSN 0740-7459. doi: 10.1109/52.43044.

[9] H. Fahmy and R. C. Holt. Software architecture transformations. In *ICSM '00: Proceedings of the International Conference on Software Maintenance (ICSM'00)*, page 88, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0753-0.

[10] S. Frey and W. Hasselbring. Model-based migration of legacy software systems into the cloud: The CloudMIG approach. In *Proc. WSR 2010*, 2010.

[11] W. Hasselbring, R. Reussner, H. Jaekel, J. Schlegelmilch, T. Teschke, and S. Krieghoff. The dublo architecture pattern for smooth migration of business information systems: An experience report. In *In Proceedings of the 26rd International Conference on Software Engeneering (ICSE-04), Los Alamitos, California, May23–28 2004. IEEE Computer Society*, pages 117–126, 2004.

[12] M. Hennig and H. Seeberger. Einführung in den extension point mechanismus von eclipse. *Java Spektrum*, 1:19 – 25, 2008.

[13] J. Izquierdo and J. Molina. An architecture-driven modernization tool for calculating metrics. *Software, IEEE*, 27(4):37 –43, jul. 2010. ISSN 0740-7459. doi: 10.1109/MS.2010.61.

[14] C. Kaner, S. Member, and W. P. Bond. Software engineering metrics: What do they measure and how do we know? In *In METRICS 2004. IEEE CS*. Press, 2004.

[15] Marcus Engelhardt, Christian Hein, Tom Ritter, Michael Wagner. Generation of Formal Model Metrics for MOF based Domain Specific Languages, 2009. URL `http://modeling-languages.com/events/OCLWorkshop2009/papers/7.pdf`. [Online; accessed 2010-09-02].

[16] T. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2:308–320, 1976. ISSN 0098-5589. doi: http://doi.ieeecomputersociety.org/10.1109/TSE.1976.233837.

[17] T. Mens and T. Tourwe. A survey of software refactoring. *IEEE Trans. Softw. Eng.*, 30(2):126–139, February 2004. ISSN 0098-5589. doi: 10.1109/TSE.2004.1265817. URL `http://dx.doi.org/10.1109/TSE.2004.1265817`.

[18] D. L. Parnas. Software aging. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press. ISBN 0-8186-5855-X.

[19] R. Reussner and W. Hasselbring, editors. *Handbuch der Software-Architektur*. dpunkt, Heidelberg, 2. edition, 2008. ISBN 978-3-89864-559-1.

[20] Santiago Comella-Dorda, Kurt Wallnau, Robert C. Seacord, John Robert. A Survey of Legacy System Modernization Approaches, April 2000.

[21] T. Systä, A. Electronica, and U. Tamperensis. Static and dynamic reverse engineering techniques for java software systems, 2000.