Damla Durmaz, Ferhat Beyaz, Sebastian Barthel, Arsenij Solovjev,
Markus Rudolph, Alexander Dümont

Sonar Modelbus Plugin
Project Manual

Softwareproject WiSe 2012
Modelbased Software Engineering Softwareentwicklung

February 17, 2013

Freie Universität Berlin, Fachbereich Informatik
AG Software Engineering, Wintersemester 2012
Ina Schieferdecker, Tom Ritter und Christian Hein

Contents

Chapter 1

Introduction

1.1 Motivation

Sonar (section 2.3) is an open source software quality platform that uses various static code analysis tools such as Checkstyle, PMD and FindBugs to extract software metrics, which can be used to improve software quality. For source code, Sonar can even give comments on analyzed code, such as what kind of major bugs are found or conventions are not hold and what can be done to resolve it. Usually, Sonar is used in combination with programming languages, like Java, C or C#, but Sonar indeed gives the functionality to display all kind of measures and handle more than just source code. From model driven software development (section 1.2) we know, that all the object oriented ideas also can be mapped to models, which represent the original objects. So in fact, we also could analyze models instead of just source code. This aspect can be very useful in enterprises, which only work model driven. Instead of transforming their models into source code, they can only rely on the standards, given for the used model language, and can analyze their models with Sonar. This has a great advantage: Instead of firstly generating code from their models and then analyzing their source code by Sonar, they simply can use the models only, analyze them with Sonar and only when the measurements reached a specified threshold, they can generate code from their models. Thus, developers can evaluate, change and improve code quality before generating code.

Within the *Softwareprojekt modellgetriebene Softwareentwicklung* at *Freie Universität Berlin*, we worked together with *Frauenhofer FOKUS*, an institution of *Frauenhofer Gesellschaft*. The corresponding *Fraunhofer-Institut für Offene Kommunikationssysteme* has a technology, called Modelbus (section 2.1), a modelling framework with a model repository behind, which is used to achieve a seamless integration between tools and the work of the development team. In Modelbus, system information (models) can be stored and reused in a transparent way. The idea is to combine the work of Sonar with the work of Modelbus: Models from the Modelbus repository can be fetched, analyzed and the results can be visualized to the developer.

Sonar metrics are metrics for normal source code. But there are a lot of Sonar metrics, which can be transferred logically to metrics for object oriented models, for example UML models. Therefore, Sonar metrics must be translated into object oriented metrics, which can be applied to models from the Modelbus repository. Since we cannot simply use all of the metrics we know from object oriented code analysis, Sonar cannot be run out of the box to analyze models. Our plugin should therefore use the Metrino service (section 2.2), which gives us the functionality to examine all of the metrics on

models as well. So with our work we want to provide a Sonar plugin to analyze models located at a Modelbus repository using the Metrino Service.

1.2 Model-Driven Engineering

With model-driven engineering we focus on domain models rather then on objects in the sense of object oriented programming. Those domain models are anbstract representations of the knowledge and activities that belong them). We can compare them with objects, but instead of describing them with sourcecode from programming languages like Java or C we simply use UML and generate the sourcecode from it.

With the usage of standardized models model-driven engineering wants to maximize the compatibility between systems and wants to increase the overall productivity. Models are developed in the communication process between the product owner, developers and other people (designers, etc). As the model set approaches completion, development process can be started.

1.3 Expectations

Our team consists of six computer science MSc students who are motivated to find a stable solution for the given task: Alexander Dümont, Arsenij Solovjev, Damla Durmaz, Ferhat Beyaz, Markus Rudolph and Sebastian Barthel.

Our goal is to expand Sonar to cooperate with Modelbus. At the beginning of the course, none of us had experience in working with neither Sonar nor Modelbus nor Metrino. So, in fact our first expectation was to learn the usage of those software systems. We also had not much experience in the model driven software developement. Therefore it should be necessary to learn to work with models and to generate code from it (as later used within the SMM parser and other software components). Although we had no experience with those systems, their good programming interfaces are promising to work fluently together.

1.4 Goal of this document

The goal of this document is mainly to provide an overview of our Sonar plugin and its architecture. In chapter 2, the used technologies for our plugin are explained briefly. Firstly, Modelbus and the Metrino service are presented. Additionally, Sonar is presented and how a Sonar plugin can be developed in general. Afterward, the architecture of our software is shown and which formats are used to communicate with the Metrino service. Finally, we want to present a problem, which raised during the development and how it can be solved. In chapter 3, an installation manual is provided and how the plugin can be used and especially, system requirements are mentioned. In the last chapter, the work will be reviewed and a conclusion is made. In the appendix (Appendix A), we included our meeting protocols, which we made during each lecturer and

other team meetings. These meeting protocols explain in detail our progress for each work, the problems, with which we were confronted and how the software progressed.

1.5 Requirements

The product owner presented us their system with Modelbus and the Metrino service. They explained, that in the Modelbus repository, there are models which can be analyzed by the Metrino service and that it can be interesting to get to know in what extend object oriented metrics of Sonar can be applied to models. Therefore, we identified some requirements for our Sonar plugin:

1. Create a plugin for sonar
2. Comprehend and cooperate with ModelBus repository
3. Transfer as many as possible metrics from Metrino to Sonar
4. Select the model
5. Traverse the directory tree
6. Analyze source codes
7. Plugin must work as efficient as possible

In the following sections, the requirements will be described in detail and are divided into functional and non-functional requirements.

Functional requirements

Create a plugin for Sonar	
Requirement Number	1
Description	Each sonar plugin has a specific architecture. The created sonar plugin must hold the specifications for a sonar plugin. Additionally, it must be run on a server and offers a WSDL interface.
Dependencies	**none** (should be done first)
Priority	high
State	closed
Fit Criterion	The plugin can be build and runs on a chosen server. A client connects to the Modelbus server and understands the WSDL interface. The client receives a message from the WSDL interface of the Modelbus server.

Create a plugin for Sonar	
Requirement Number	1
Description	Each sonar plugin has a specific architecture. The created sonar plugin must hold the specifications for a sonar plugin. Additionally, it must be run on a server and offers a WSDL interface.
Dependencies	**none** (should be done first)
Priority	high
State	closed
Fit Criterion	The plugin can be build and runs on a chosen server. A client connects to the Modelbus server and understands the WSDL interface. The client receives a message from the WSDL interface of the Modelbus server.

Comprehend and cooperate with ModelBus repository	
Requirement Number	2
Description	ModelBus has an own repository, where it stores artifacts of the tools, which are connected to the ModelBus via adapters. It must be understood, how ModelBus connects to the repository and how an external tool (like a sonar plugin) can do this.
Dependencies	# 1
Priority	high
State	closed
Fit Criterion	The sonar plugin is able to connect to the ModelBus repository via a command. It can fetch the source code for a random project to test, if the connection is established. The sonar plugin can terminate the connection to the repository for a given command.

Select the model	
Requirement Number	4
Description	The sonar plugin can fetch model code from the repository and transform the model into a format which can be read by model metric analyzing tools, like Metrino.
Dependencies	# 3
Priority	high
State	closed

Fit Criterion	The plugin fetched a model. It transforms the model into the format of Metrino. For a test case, the transformed model shall be saved in a file and read successfully by Metrino.

Traverse the directory tree	
Requirement Number	5
Description	In the directory tree of sonar plugin, a lot of source code and model code projects are placed. The plugin should be able to analyze all codes from the repository.
Dependencies	**none**
Priority	high
State	closed
Fit Criterion	The plugin fetched the whole project tree of the repository. It loads each project into its own directory tree. It traverses the whole directory tree. For each project in its directory tree, it analyzes the code and visualizes it on the web interface.

Analyze source codes	
Requirement Number	6
Description	The sonar plugin can apply a set of metrics to the fetched source code. The results are visualized with graphs or statistics and can be get via the web interface.
Dependencies	# 2
Priority	high
State	Canceled, because requirement is unfeasible with current Sonar version (could be with future versions)
Fit Criterion	The plugin fetches source code from the repository. It applies a couple of metrics, which can be chosen from the application interface.

Non-functional requirements

Transfer as many as possible metrics from Metrino to Sonar	
Requirement Number	3

Description	The sonar plugin can already fetch model code from the repository. The models are read by Metrino and analyzed by Metrino with object oriented metrics. The results of the analysis are fetched from Metrino by the sonar plugin and visualized on the web interface. Metrino offers a lot of metrics and it should be possible to offer as many metrics from Metrino as possible.
Dependencies	# 4
Priority	high
State	open
Fit Criterion	The sonar plugin sends a model to Metrino with the information which object oriented metric shall be applied. Metrino returns the results. The results are visualized. The plugin offers to select at least 10 object oriented metrics of Metrino to apply to the sent model code.

Plugin must work as efficient as possible	
Requirement Number	7
Description	Plugin must traverse the directory tree fastly. The connection to the ModelBus must provide security standards, but it should not be slow. The results of the analysis should be offerred via the web interface as fast as possible.
Dependencies	# 6
Priority	middle
State	closed
Fit Criterion	Because of the difficulty of the decision, if it works properly fast, a usability test with the customer must be done, who can tell in a better way, if the product works fast or not.

1.6 Responsibilities and Progress

In this chapter, we present the project planning during the lectures. The table shall show, to which time each group member had what kind of tasks, how long it took to solve the task and what kind of tasks raised. The figure shows the tasks over the time and the responsible person.

Exercise	**Begin**	**End**	**Progress**	**Done By**
Explore Modelbus	Mo 05.11.12	Fr 09.11.12	100%	Team
Explore Sonar	Mo 05.11.12	Fr 09.11.12	100%	Team

Explore plugin developement possibilities	Mo 05.11.12	Fr 09.11.12	100%	Ferhat
Explore Sonar features and developement possibilities	Mo 05.11.12	Fr 09.11.12	100%	Damla
First attempt to plugin developement	Mo 05.11.12	Fr 09.11.12	100%	Markus
Wiki Introduction	Mo 05.11.12	Fr 09.11.12	100%	Sebastian
Modelbus connection	Mo 05.11.12	Fr 09.11.12	100%	Arsenij
Sonar documentation	Mo 05.11.12	Fr 09.11.12	100%	Markus
Wiki Metrino	Mo 05.11.12	Fr 09.11.12	100%	Alexander
Sonar plugin documentation	Mo 05.11.12	Fr 09.11.12	100%	Ferhat
Modelbus documentation	Mo 05.11.12	Fr 09.11.12	100%	Damla
Requirements	Mo 05.11.12	Fr 09.11.12	100%	Damla & Ferhat
Presentation Requirements	Mo 05.11.12	Fr 09.11.12	100%	Ferhat
Improve requirements	Mo 12.11.12	Fr 16.11.12	100%	Damla
Architecture conecept for sonar	Mo 12.11.12	Fr 16.11.12	100%	Ferhat
Architecture conecept for metrino	Mo 12.11.12	Fr 16.11.12	100%	Alexander
Introduction to wiki	Mo 12.11.12	Mo 19.11.12	100%	Sebastian
Presentation Template	Mo 12.11.12	Fr 19.11.12	100%	Damla
Software architecture – sequencediagram/activitydiagram	Mo 12.11.12	Fr 23.11.12	100%	Sebastian
Setup Modelbus Repository Server	Mo 19.11.12	Mo 26.11.12	100%	Arsenij
Explore howto running sonar on a repository	Di 20.11.12	Mo 26.11.12	100%	Arsenij
Installation of software components	Do 23.11.12	Do 23.11.12	100%	Team
Example Sonar plugin	Mo 19.11.12	Fr 23.11.12	100%	Unknown
Multilanguage (different programming languages) support in sonar plugins	Fr 23.11.12	Do 29.11.12	100%	Markus & Damla & Ferhat
Sequence diagram for the work of sonar with Metrino and ModelBus	So 25.11.12	So 25.11.12	100%	Damla & Sebastian
Setup Modelbus server	Fr 23.11.12	Mo 26.11.12	100%	Markus
Presentation first architecture	Fr 23.11.12	Mo 26.11.12	100%	Ferhat
Activity diagram modelbus repository	Fr 23.11.12	Mo 26.11.12	100%	Damla
SOAP Client Metrino	Fr 23.11.12	Do 06.12.12	100%	Alexander
Wiki Logo and header	Mo 26.11.12	Fr 30.11.12	100%	Ferhat
OCL & Documentation	Mo 26.11.12	Fr 30.11.12	100%	Damla & Ferhat

Milestone definitions and description	Mo 26.11.12	Fr 30.11.12	100%	Sebastian
Installation manual	Mo 03.12.12	Mo 10.12.12	100%	Sebastian
Include other language plugins with modules	Mo 03.12.12	Mo 10.12.12	100%	Arsenij
Client without using WSDL by hand	Mo 03.12.12	Mo 10.12.12	100%	Markus
Objectoriented Analysis to Models	Mo 03.12.12	Mo 10.12.12	100%	Ferhat
Understand Metrino metrcis	Mo 10.12.12	Fr 21.12.12	100%	Sebastian & Alexander
Download modelbus files in a Sonar plugin	Mo 10.12.12	Fr 21.12.12	100%	Markus
Create latex documentation	Mo 17.12.12	Fr 21.12.12	100%	Ferhat
Meeting Protocols to latex documentation	Mo 17.12.12	Fr 21.12.12	100%	Damla
Metrino CheckModel, download and parse SMM from Repo	Mo 07.01.13	Mo 21.01.13	100%	Arsenij
Sonar Frontend	Mo 07.01.13	Mo 21.01.13	100%	Sebastian
Parser for SMM	Mo 14.01.13	So 20.01.13	100%	Damla & Ferhat
Using EMF to parse SMM files	Mo 21.01.13	Mo 28.01.13	100%	Damla & Ferhat
Include parser into project workflow	Mo 21.01.13	Mo 28.01.13	100%	Damla & Ferhat
Merge all components into one	Mo 21.01.13	Mo 28.01.13	100%	Arsenij
Project spezific measurements in sonar	Mo 21.01.13	Mo 28.01.13	100%	Markus
Ceate OCL metrics	Mo 21.01.13	So 03.02.13	100%	Alexander
Color Measurements	Mo 21.01.13	Mo 28.01.13	100%	Sebastian
Documentation in Latex	Mo 28.01.13	Mo 18.02.13	25%	Team
SMM Adapter	Mo 28.01.13	Mo 18.02.13	67%	Arsenij
Load dynamic metrics in sonar	Mo 28.01.13	Mo 18.02.13	36%	Sebastian
Meeting Protocols	Mo 03.12.12	Mo 18.02.13	95%	Damla & Ferhat

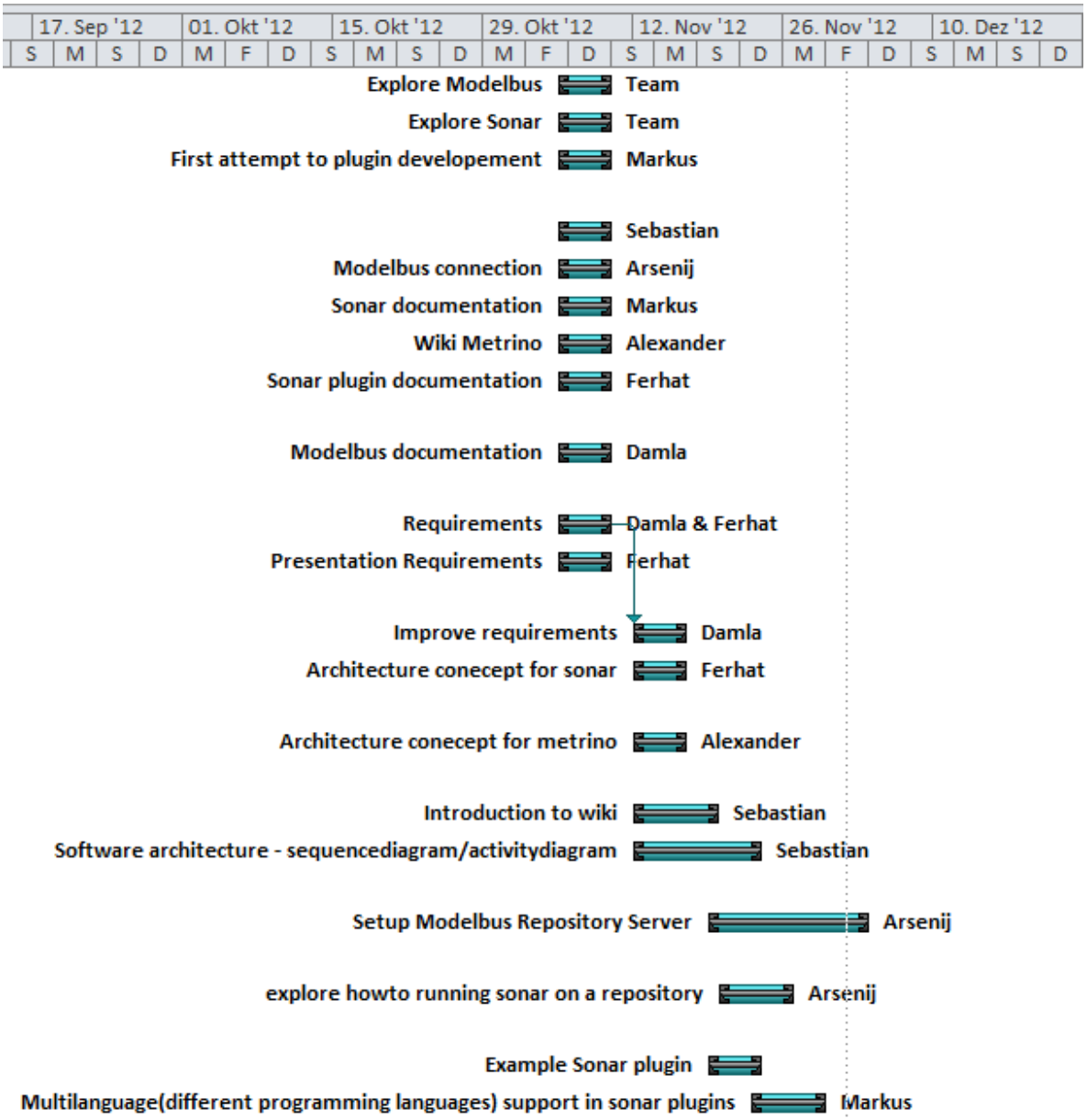

Figure 1.1: Exercises - MS Project screenshot 1

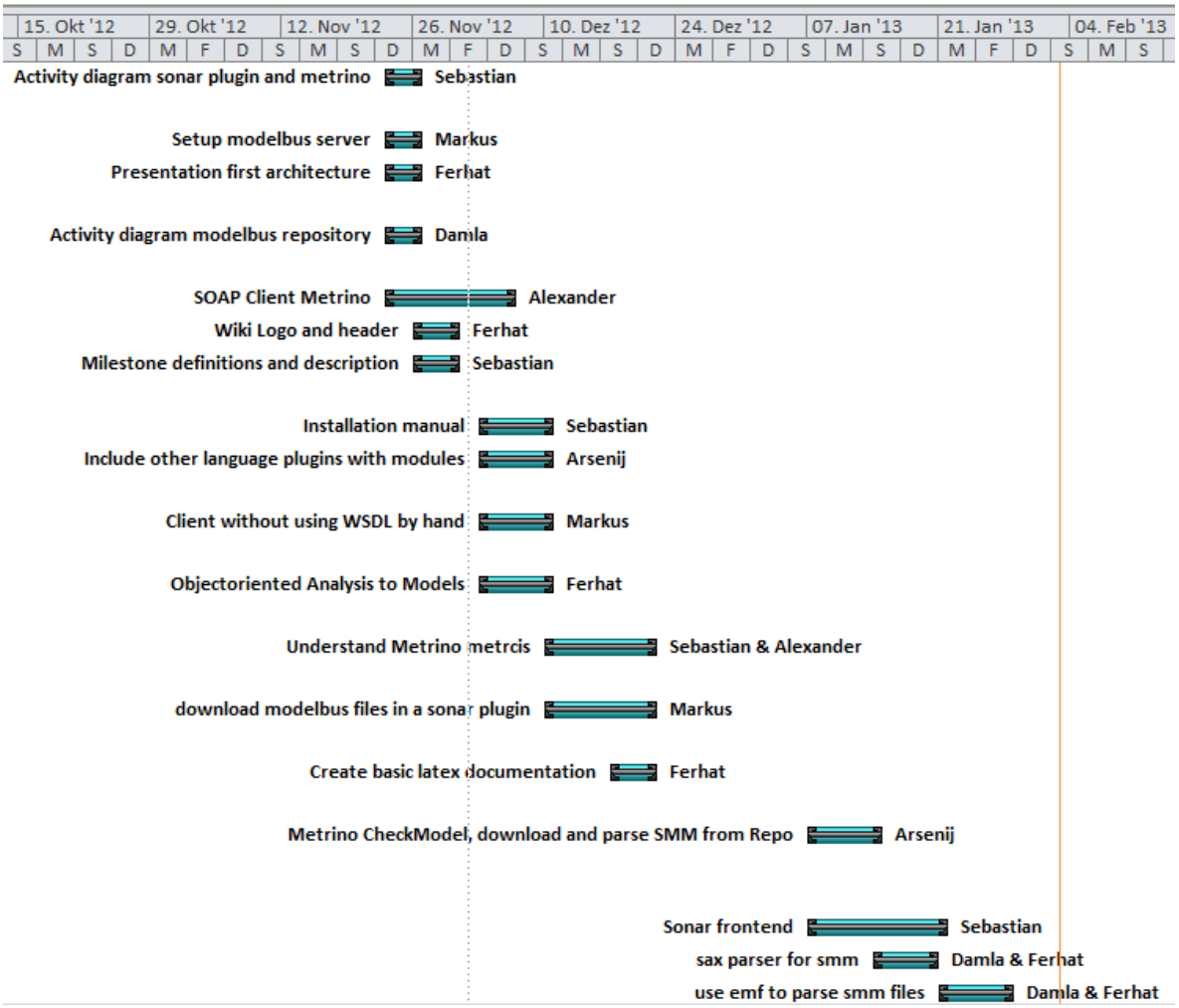

Figure 1.2: Exercises - MS Project screenshot 2

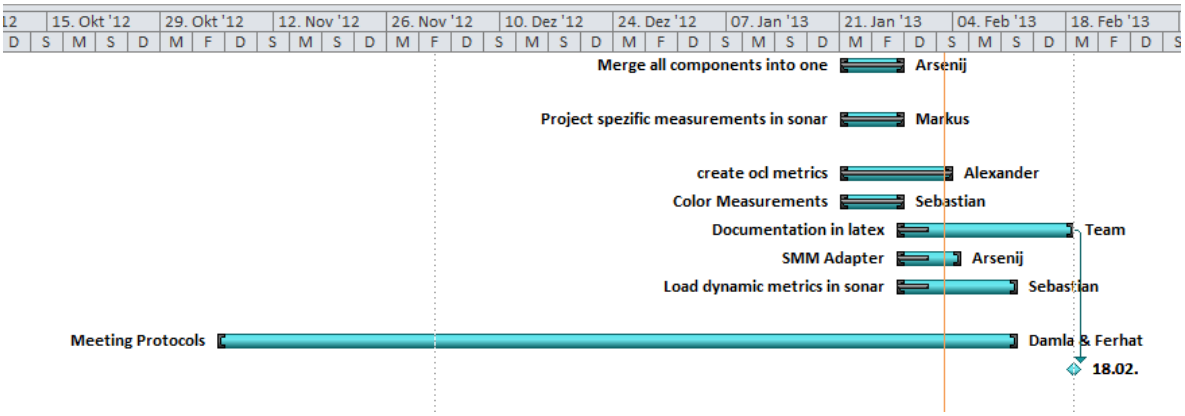

Figure 1.3: Exercises - MS Project screenshot 3

Chapter 2

Sonar Modelbus Plugin

2.1 Modelbus

Scenario

Imagine, you are in a software developement process. You and your team partners use different developement tools and a lot of artefacts are produced by different team members. In such a case, the artefacts may be inconsitent and a developer needs to make updates and changes by hand to keep them consistent. A tool which would automate this process and do more, is ModelBus.

What is ModelBus?

It is a model-driven open source framework for the integration of development tools during a MDE process. It keeps the artefacts of the development process consistent. It offers a communication between tools with the help of adapters: the tools are connected to the bus via adapters and can offer their services to other tools connected to the bus.

Goals

- Data Integration: Tools can share (data) models
- Control Integration: Tools connected to the bus can use the service of other
- connected tools
- Process Integration: Several tools are used together in the development and are highly supported
- Support: Support of distributed MDE development
- Architecture: Based on SOA
- Architecture
- based on SOA
- central bus-like communication structure number of core services

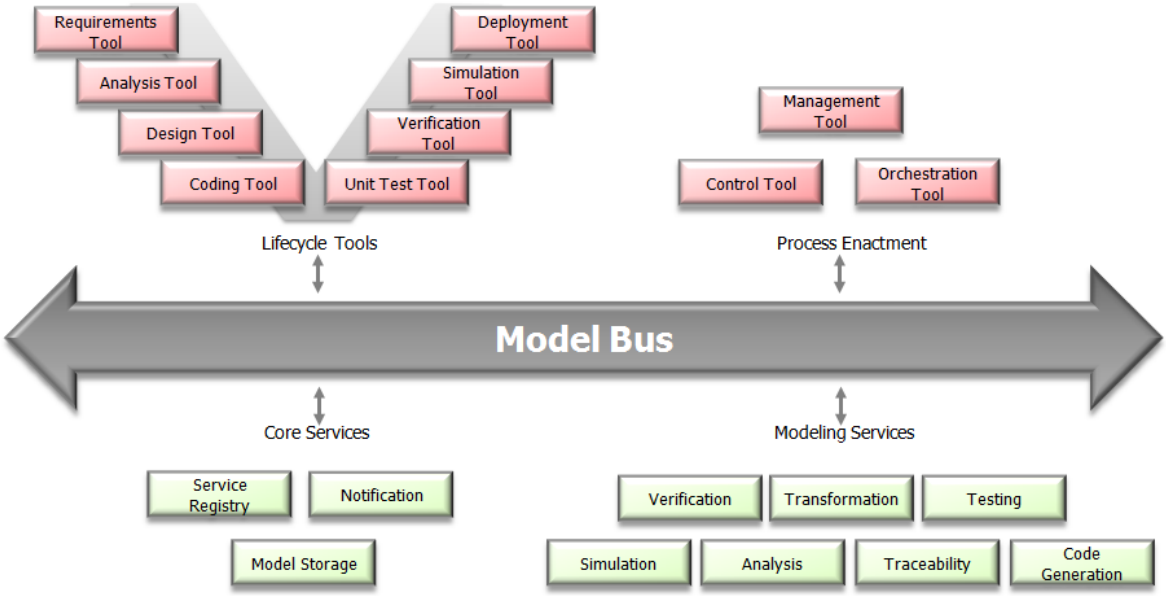

Figure 2.1: Modelbus tools and services

- a copule of model management tools
- different tools can be added to the bus via adapters
- if a tool is connected, it is handled as a service for other tools orchestration: a lot of tools (with their service) and automation are connected
- together to a complex system

Features

Automation of development tasks

- define tasks as modeling services
- orchestrate defined tasks with other modelling services
- orchestrations can be run automatically (by user or other orchestrations)

Inbuilt and transparent model management

- problem: development produces artefacts and developer don't want to care about where they came from and how they have to be imported
- artefacts are acceessed to the rigth model by each connected tool on the ModelBus bus

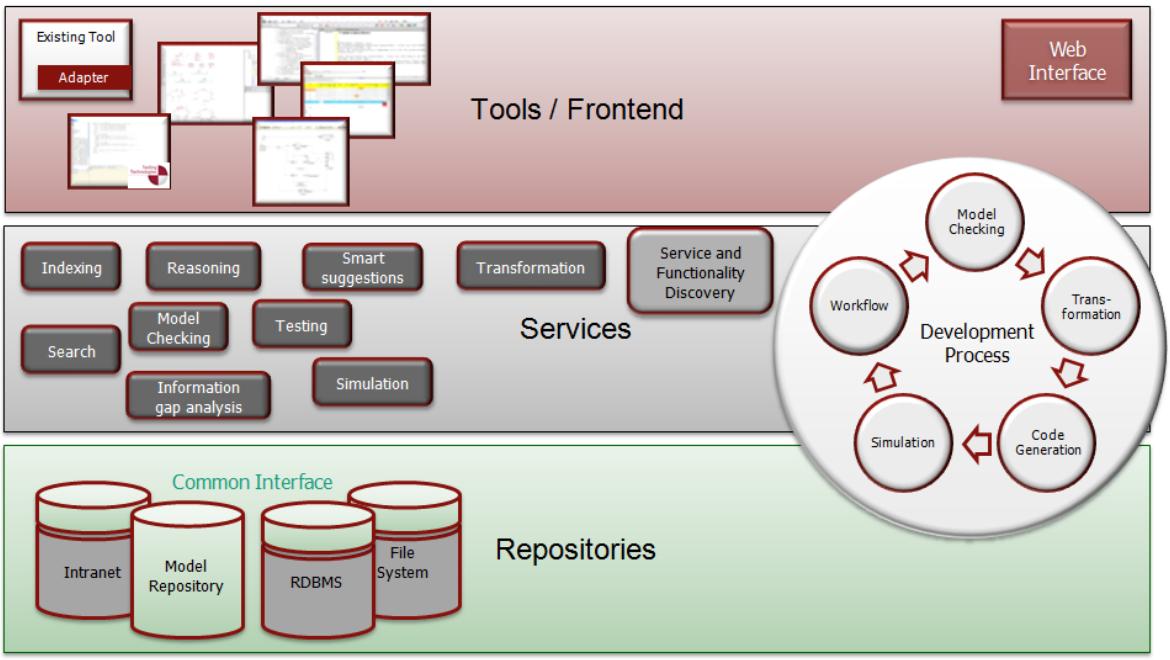

Figure 2.2: Developement process and layers

Support of large and complex models

- models often have complex structure
- model A imports parts if model B and B references package C and ...
- thus, a lof ot developers work on the models concurrently
- solution: versioning of (complex) models and model fragments in the model repository
- core service notification: notification system, which informs about changes to a model

Distributed and heterogeneous

- normally, if a tool is updated, different artefacts may be non-compatible together
- this may happen frequently in larger development teams
- but this would be crucial, espepcially for model operations tools (for example tools, which transform models)
- solution: via the adapters, models from the tools are translated into a ModelBus known format

- distribution of models is therefore realized
- data models stay consistent, even when tools are updated

Built on industry standards

- Transportation: HTTP, HTTPS, XMPP, CXF, JMS, SOAP
- Core services: Distributed OSGi, SVN, EMF
- ...

Adapters for tools

- Doors
- Eclipse Papyrus
- Enterprise Architect
- Eclipse TeamProvider
- Office
- Rational Software Architect
- Rhapsody
- Simulink

2.2 Metrino

Short facts

- Validating models by information quantity and quality
- Uses OCL (Object Constraint Language) and SMM (Software Metrics Metamodel)
- Can be used as stand-alone tool or as ModelBus add-on with additional service features
- Works as a set of support tools in four conceptual phases
- Check of these selectable guidelines result in validation, warnings and/or errors
- Handles UML models, any Domain Specific Modeling Language (DSL) based on MOF

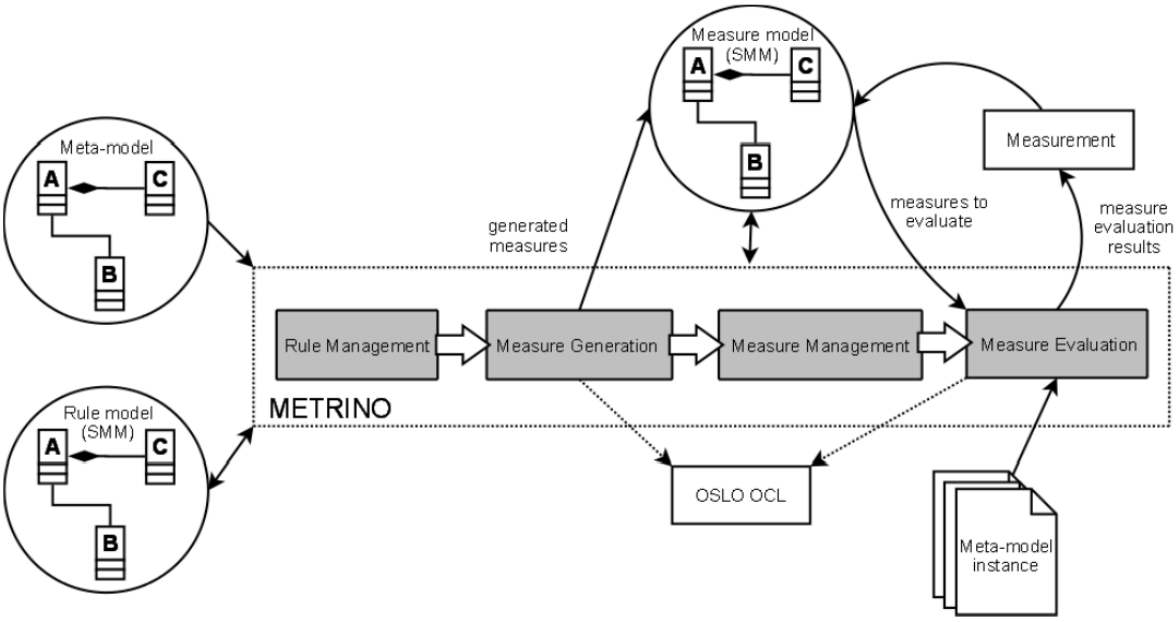

Figure 2.3: Developement process and layers

- Manage and compute generated or user-defined, domain specific measures
- Already includes a set of metrics in the current version
- Supports customizable report generation to different formats
- Supports visualization of computational results, e.g. graphs
- Installation by Eclipse update-site through ModelBus website: http://www.modelbus.org/metrino/

Four phases

More

- Slides of the presentation given at the OCL Workshop in Denver: Generation of Formal Model Metrics for MOF based Domain Specific Models (Marcus Engelhardt, Christian Hein, Tom Ritter, Michael Wagner)
- Christian Hein, Marcus Engelhardt, Tom Ritter and Michael Wagner: Generation of Formal Model Metrics for MOF based Domain Specific Languages, OCL 2009 Workshop at ACM/IEEE Models 09 Conference, USA, September 5th 2009

Screencasts

Worth seeing, brief presentations about usability and features of Metrino:

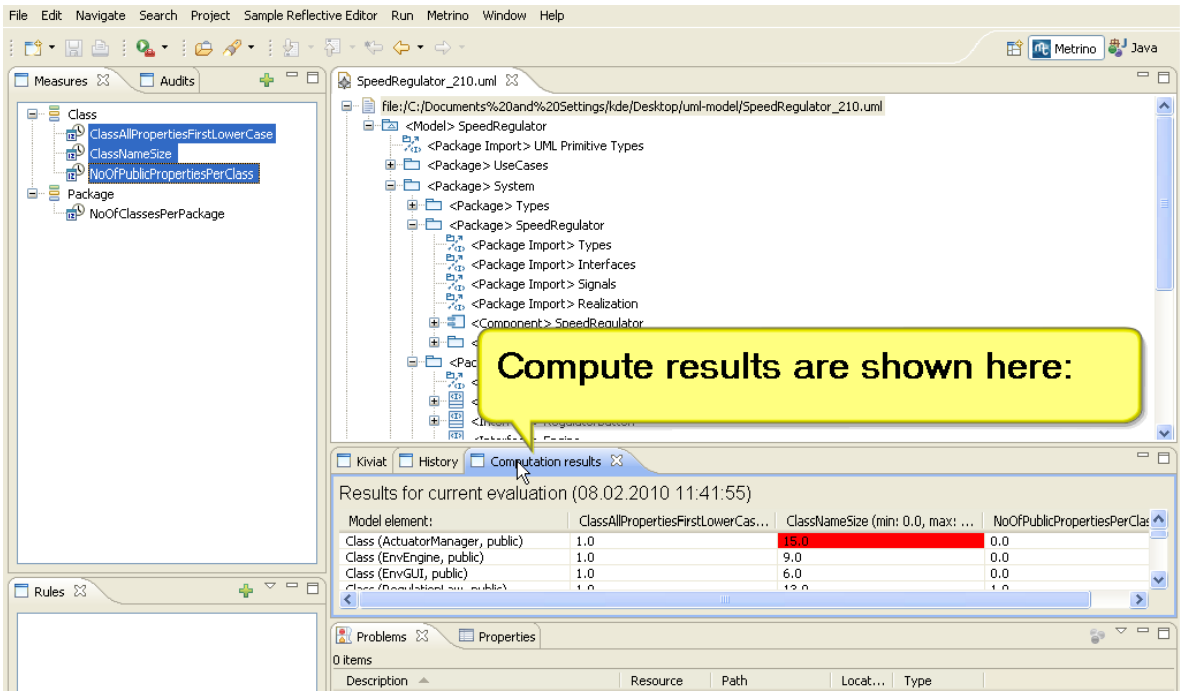

Figure 2.4: Developement process and layers

- Short screencast showing the usage of Metrino for UML Models - The screen cast shows the evaluation of "hand coded" measures on a UML Model representing the results as tables and KIVIAT graphs

- Screencast of a Metrino Validation - The screen cast shows the generation and evaluation measures on for a DSL based model and the integration into validation framework.

Problems

The installation of the current version of Metrino is not possible due to malformed dependencies. Unfortunately a recursively installing of the required dependencies does not help, since they need other dependencies by themselves, resulting in an infeasible search-and-install marathon. Screenshot

Screenshots

These are only a subset of interesting screenshots taken from the screencasts above:
(generic validation by selected measures)
(graphical representation of a selected measurement)
(rules, measures, problems and validation)

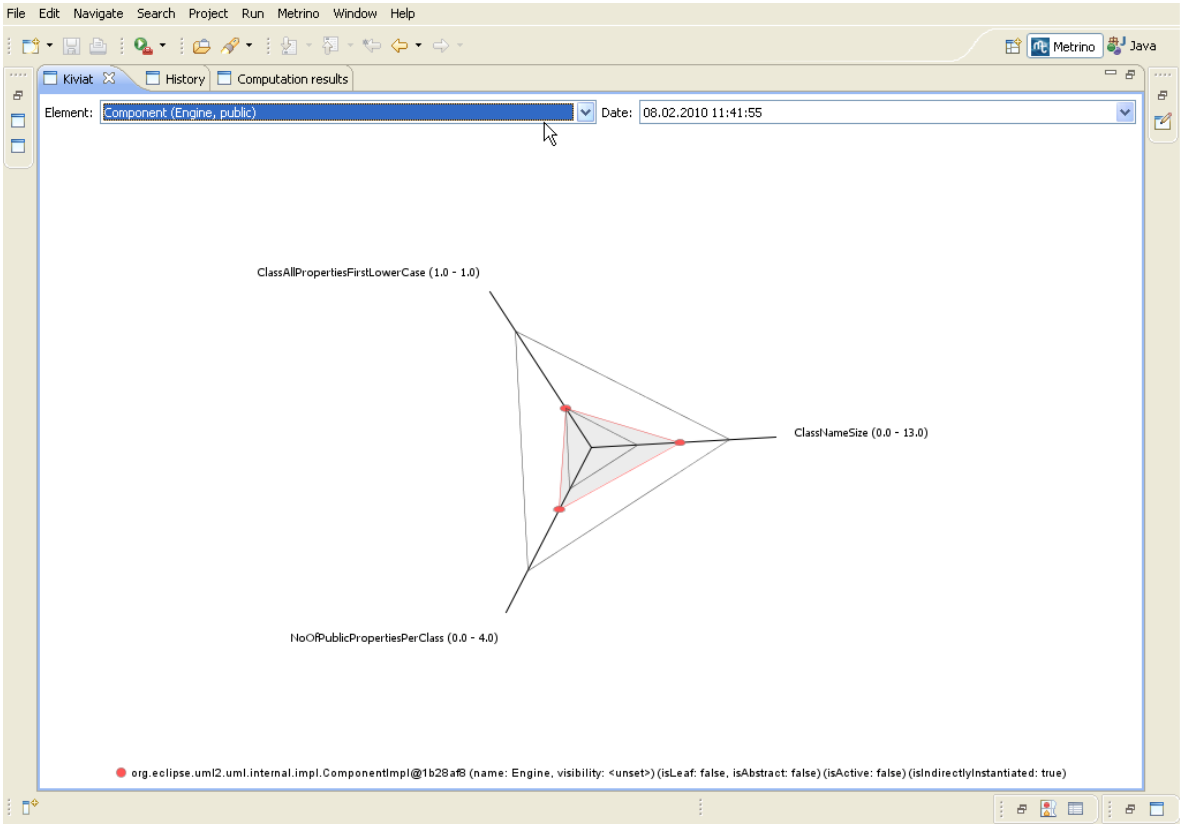

Figure 2.5: Developement process and layers

Declaration of OCL rules

With the models, we got from the Modelbus service we can start an analysis. Therefore we want to use Metrino, which expects OCL (Object Constraint Language) as a description of the requests. OCL generally spoken is a declarative language for describing rules which apply on UML-Models.

In OCL there are 7 constraints to distinguish: 1. Invariants have to apply on either an instance or an association. 2. Pre- and postconditions have to apply every time, when the according operation begins or ends. 3. Initial and derived values are the constraints for other derived values. 4. You can define new attributes and operations, that are not defined in the model. 5. If there is a state transition, guards have to apply.

With this set of rules, we can define metrics on the models.

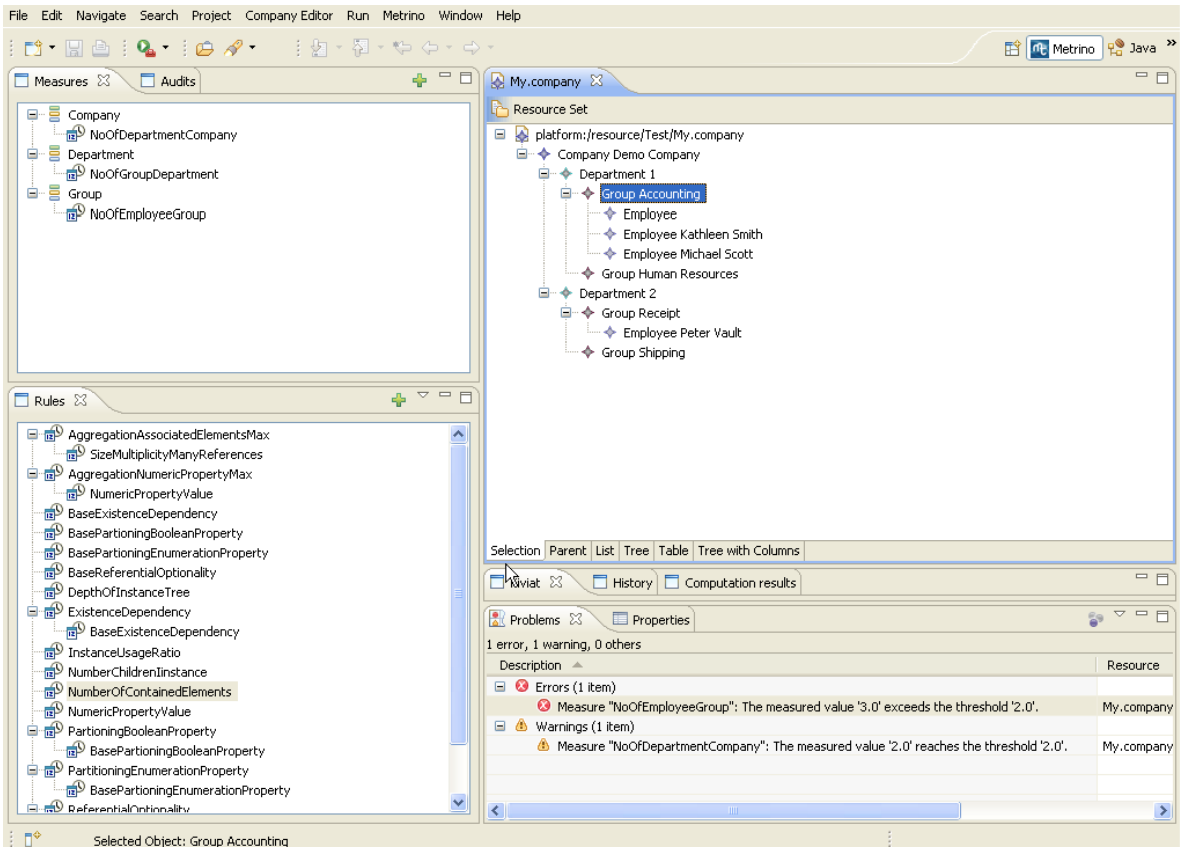

Figure 2.6: Developement process and layers

OCL

2.3 Sonar

Sonar is an open platform to manage code quality. It offers reports on duplicated code, coding standards, unit tests, code coverage, complex code, potential bugs, comments and design and architecture.

Sonar consists of 3 components:

- A database that stores the configuration and results of quality analysis
- A web server that is used to navigate the results of the analyzes and make configuration
- A client that will run source code analyzers to compute data on projects Covering new languages, adding rules engines, computing advanced metrics can be done through a powerful extension mechanism. More than 50 plugins are already available.

Primary supported language is Java. Other languages are supported with extensions. Several open source and commercial extensions can cover the following languages: C, C#, PHP, Flex, Groovy, JavaScript, Python, PL/SQL, COBOL and Visual Basic 6.

It integrates with Maven, Ant and continuous integration tools (Atlassian Bamboo, Jenkins, Hudson).

2.4 Developing Sonar Plugins

Building and Packaging

To create a plugin for Sonar, at least Java 5 is needed. Maven is also required to compile and package the plugin. The documentation of Sonar says, that a recommanded way is to duplicate one of the example plugins, which can be found in the /plugins directory of the github repository: https://github.com/SonarSource/sonar-examples

It is recommanded to use /plugins/sonar-reference-plugin. The example plugin can be copied by cloning the repository. Github provides the possibility to download the repository directly.

The plugin can be built and deployed by executing in the plugin root directory, so for example /path/to/sonar-reference-plugin:

```
mvn clean install
```

Thereafter a JAR file is generated in the /target directory. After copying this JAR to the /extensions/plugin directory of Sonar, the server has to be restarded. There you go, you packaged your own Sonar plugin.

Creating an own plugin

A Sonar plugin is a set of Java objects, which implement extension points, which are interfaces or abstract classes to model an aspect of the system and define contracts of what needs to be implemented. Such extensions could be pages in the web application or sensors generating measures.

This plugin extensions must be declared in a Java class, that extends org.sonar.api.Plugin. This class must then be declared in the pom with the property :

```
<artifactId>sonar-foo-plugin</artifactId>
<packaging>sonar-plugin</packaging>
<build>
    <plugins>
        <plugin>
            <groupId>org.codehaus.sonar</groupId>
            <artifactId>sonar-packaging-maven-plugin</artifactId>
            <version>1.1</version>
            <extensions>true</extensions>
```

```
            <configuration>
                <pluginClass>com.mycompany.sonar.MyPlugin</pluginClass>
            </configuration>
        </plugin>
    </plugins>
</build>
```

There are also more advanced parameters like Maven descriptors, etc. possible. The full list of advanced parameters can be found here.

There is a list of all known sub-interfaces and implementing classes of org.sonar.api.Extension. The most important and well known extension points are listed here.

Sensors and Decorators

Two extension enable methods to save measures: sensors and decorators. In plugin development it is often a problem to decide which one to use.

Sensor

A sensor is invoked once during the analysis of the project. The sensor then can invoke a maven plugin, parse flat files or connect to web servers. The generated XML file is parsed and used to save the first-level of measures on resources (project, package or class). The sensor can access and save measures on the whole tree of resources. They generally are used to add measures at the lowest level of the resource tree.

Decorator

Decorators are used when all sensors have completed their work. The decorate method is called on every resource of a certain level bottom up. Decorators load (SELECT) and save (INSERT) measures. Because of contextual calls it is only possible to access the resource and its children. So decorators are generally used to consolidate measured at higher levels that have been added by sensors at lower levels.

Modify the front-end of Sonar and working with sensors and decorators

Our plugin front-end consists of the following structure:

Package edu.swp.modelbus.reference

ModelbusMetrics.java

Our class ModelsbusMetrics implements the Metrics Interface of Sonar. In this file we define all Metrics that we'll create in other files. The obligatory metric definitions consist of the name, key and the return type. Optionally you can add more parameters with the description, direction, setQualitativ, and setDomain methods. At least we have

to define the method getMetrics() because of the interface Metrics. In this method we add all metrics to an array list and set this as return value of the getMetrics() method.

ModelbusPlugin.java

This file is the main entry point for Sonar. With annotations we can define some Plugin meta data like the description and the name. The class ModelbusPlugin extends the class SonarPlugin and has the method getExtensions(). We add in this method all Sonar extensions that we want to use like definition classes. batch classes or ui classes.

Package edu.swp.modelbus.batch

ExampleSensor.java

We created an exampleSensor to demonstrate how it must be defined. In this file it's possible to do all things you like to do. An example would be to connect to a server. In our case we don't need a sensor.

RandomDecorator.java

Our first decorator is a real random decorator. It will give a random value to all files.

Package edu.swp.modelbus.ui

ExampleFooter.java

It's possible to change the layout with plugins. This file implements the interface Footer. We just add the method getHtml() and return an example text to change the front-end layout of sonar.

ExampleRubyWidget.java

We can also add Widgets. The administrator can add widgets to the users layout. If our sensors get some values, the can be written from our widget and presented with plain text or with visual objects.

2.5 Software Architecture

The main architecture is described as the following. Our main software product is called the Sonar ModelBus Plugin which of course is a Sonar Plugin. This plugin, the project is dedicated to, implements a Metrino Adapter as well as a ModelBus Adapter. These sub programs are intended to communicate with the WebServices from Metrino and ModelBus.

(These diagrams are currently in work and not considered final.)

Simple Architecture / Package Diagram

Ecore-Diagramm

2.6 Metrics Input File SMM

SMM distinguishes between measures as the evaluation process of particular quality aspects of software artifacts and measurements which can be interpreted as the results

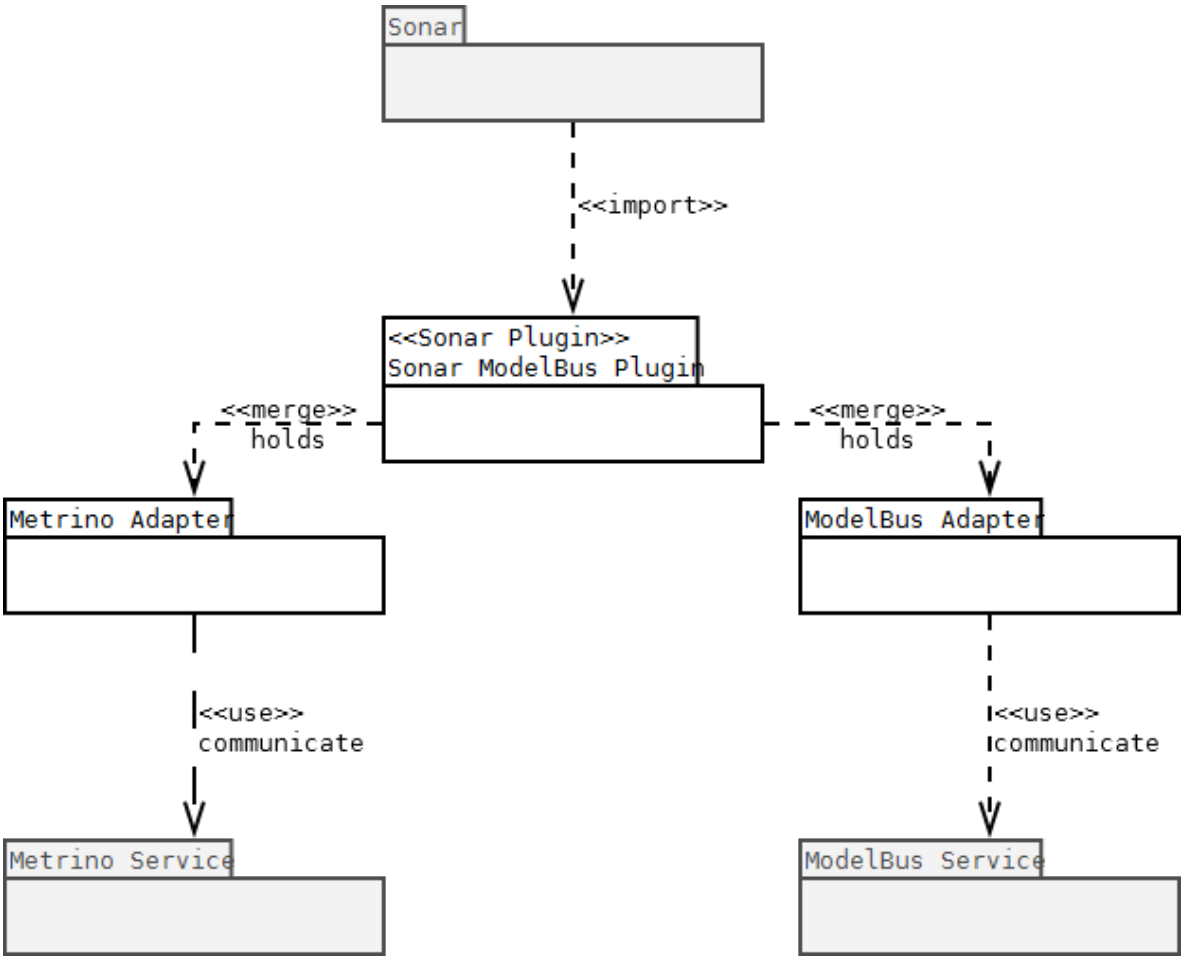

Figure 2.7: Sonar plugin package diagram

of those processes. SMM specifies several types of measures and measurements for different outcome values:

- DimensionalMeasure
- DirectMeasure
- BinaryMeasure
- CollectiveMeasure

The measures are calculated in a given scope, which has to be specified with *scope="x"*, where *x* is the id of a scope element, specified with *xsi:type="SoftwareMetricsMetamodel2:Sco* Every measure also is part of a category, therefore it has to define its category with *category="x"*, where *x* is the id of a category element, specified with *xsi:type="SoftwareMetricsMetamodel2:S* This way measures can be categorized, so that one can measure different metrics in the

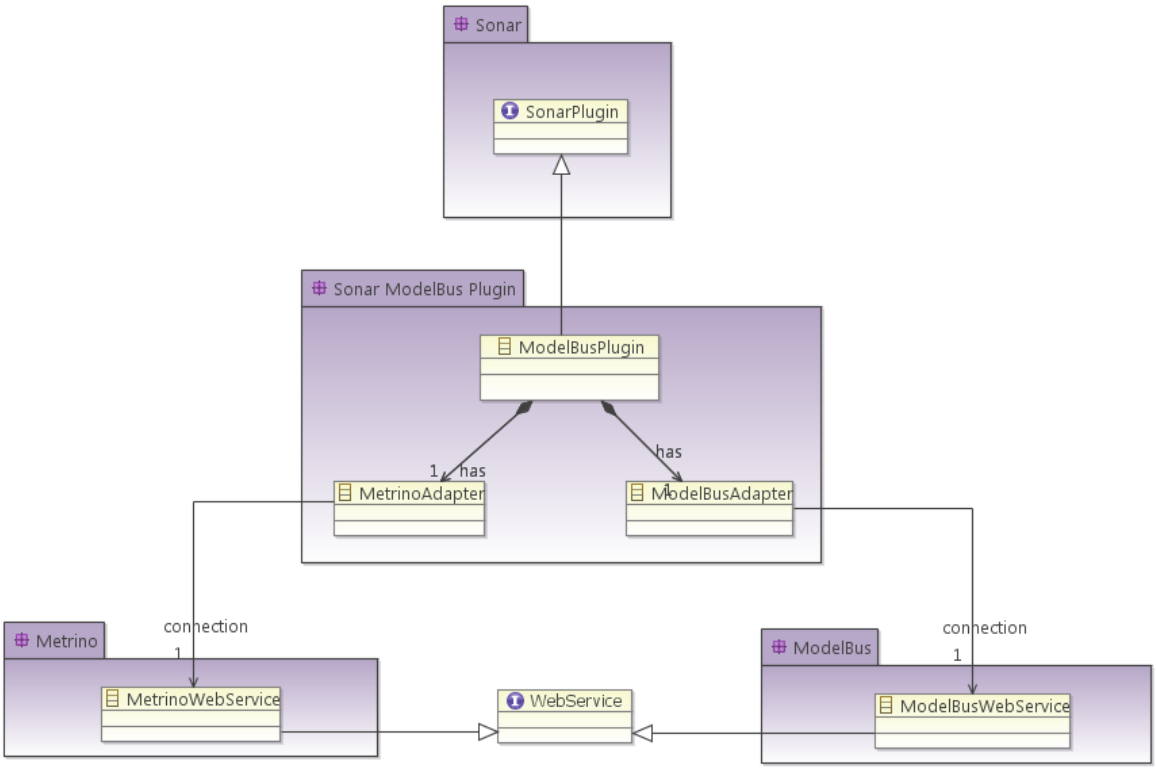

Figure 2.8: Sonar plugin ecore diagram

same SMM file. Every category has an attribute measureElement, which simply holds the ids of the measures in the category. The results of the measures are saved in a measurement with *xsi:type="SoftwareMetricsMetamodel2:DirectMeasurement"*. The id of this element has to be saved in the measurement attribut in the measure.

For more information, see Software Metrics Metamodel (SMM) on Wikipedia.

The following code is an examplary SMM file:

```
<?xml version="1.0" encoding="UTF-8"?>
<SoftwareMetricsMetamodel2:SMMModel xmi:version="2.0" xmlns:xmi="http://www.omg.org/XM
    <sMMElement xsi:type="SoftwareMetricsMetamodel2:SMMCategory" xmi:id="_yBNwwEg1EeKr
    <sMMElement xsi:type="SoftwareMetricsMetamodel2:DirectMeasure" xmi:id="_YrZQoUg4Ee
    <sMMElement xsi:type="SoftwareMetricsMetamodel2:Scope" xmi:id="_m25GAEg4EeKu7IXnh6
    <sMMElement xsi:type="SoftwareMetricsMetamodel2:DirectMeasure" xmi:id="_w9WYoEg4Ee
    <sMMElement xsi:type="SoftwareMetricsMetamodel2:DirectMeasurement" xmi:id="_CkmbOE
        <measurand href="model/MetrinoSampleModel.uml#_QYmMoEg7EeKe9JB-jPya7w"/>
    </sMMElement>
    <sMMElement xsi:type="SoftwareMetricsMetamodel2:Observation" xmi:id="_CkmbOUg8EeKe
        <whenObserved xmi:id="_CkmbOkg8EeKe9JB-jPya7w" value="1355743348117"/>
    </sMMElement>
```

```
    <sMMElement xsi:type="SoftwareMetricsMetamodel2:DirectMeasurement" xmi:id="_CksicE
        <measurand href="model/MetrinoSampleModel.uml#_QYmMoEg7EeKe9JB-jPya7w"/>
    </sMMElement>
    <sMMElement xsi:type="SoftwareMetricsMetamodel2:Observation" xmi:id="_CksicUg8EeKe
        <whenObserved xmi:id="_Cksickg8EeKe9JB-jPya7w" value="1355743348117"/>
    </sMMElement>
    <sMMElement xsi:type="SoftwareMetricsMetamodel2:DirectMeasurement" xmi:id="_NDgqsE
        <measurand href="model/MetrinoSampleModel.uml#_QYmMoEg7EeKe9JB-jPya7w"/>
    </sMMElement>
    <sMMElement xsi:type="SoftwareMetricsMetamodel2:Observation" xmi:id="_NDgqsUg8EeKe
        <whenObserved xmi:id="_NDgqskg8EeKe9JB-jPya7w" value="1355743418577"/>
    </sMMElement>
    <sMMElement xsi:type="SoftwareMetricsMetamodel2:DirectMeasurement" xmi:id="_NDmxUE
        <measurand href="model/MetrinoSampleModel.uml#_QYmMoEg7EeKe9JB-jPya7w"/>
    </sMMElement>
    <sMMElement xsi:type="SoftwareMetricsMetamodel2:Observation" xmi:id="_NDmxUUg8EeKe
        <whenObserved xmi:id="_NDmxUkg8EeKe9JB-jPya7w" value="1355743418577"/>
    </sMMElement>
</SoftwareMetricsMetamodel2:SMMModel>
```

2.7 ModelBus Client Tool

We created a tool for checking in and out files from/to the ModelBus repository. It can be simply called via "make":

```
Use
  make checkin URI=http://uri.de/location/in/repository/file.txt FILENAME=location/to/
to checkin a file located at FILENAME into the repository at a position defined by the

Use
  make checkout URI=http://uri.de/location/in/repository/file.txt FILENAME=location/to
```

to checkout a file from the repository located at URI to a local position defined by the FILENAME. Changes can be made at the Client.java file. You can compile it by calling "make install". This will compile and assemble the client with its ModelBus dependencies.

2.8 ClassLoader Problem

The problem was that we could not connect to the ModelBus repository through the sonar plugin. Using the same code in a standalone Java application instead worked fine: The ModelBus repository was successfully checked out.

The reason was that the ModelBus part could not resolve some bindings in its configuration. That was caused by the missing of a resource. This resource lied in the META-INF folder of the "org.modelbus.cxf.dosgi" jar bundle. The loading of this resource was done over the current context class loader in a ModelBus class:

```
ClassLoader cl = Thread.currentThread().getContextClassLoader();
Enumeration<URL> urls = cl.getResources("META-INF/cxf/bus/bus-extensions.txt");
//loading each url...
```

The problem is that the context classloader of the current thread is not the plugin classloader.

Each Sonar plugin runs in an isolated classloader to avoid conflicts with other plugins. Third-party libraries can be loaded by using a mechanism specific to Sonar. It only requires to build the plugin with the sonar-packaging-maven-plugin, which copies the libs into META-INF/lib. This mechanism relies on its own classloader implementation.

By temporarly replacing the context class loader with the plugin class loader (the class loader of the extension), this issue can be solved:

```
ClassLoader initialClassLoader = Thread.currentThread().getContextClassLoader();
try {
  Thread.currentThread().setContextClassLoader(getClass().getClassLoader());
  //access ModelBus...
} finally {
  Thread.currentThread().setContextClassLoader(initialClassLoader);
}
```

Chapter 3

Installation and Usage

3.1 Software Requirements

For development following software components are needed:

- Java (JDK) 1.6 or 1.7
- Maven
- Eclipse (Juno)
- Maven Plugin for Eclipse
- Git or Egit for Eclipse
- Modelbus Team Provider for Eclipse
- Modelbus 1.9.7
- Metrino for Modelbus with following plugins
  - Metrino rule evaluator plugin
  - Metrino measure plugin

Sonar will be downloaded and deployed via Maven at compile time.

3.2 Installation Manual

In the following sections we explain every detail. Just skip a section if you think you still have done that what the regarding section focuses.

Java

We tested the software with Java 1.6 and 1.7. You can get those from here. Select and download the current JDK to install it on your system. If Java is ready, make sure that the following system variables exist:

```
JAVA_HOME = C:\Program Files\Java\jdk1.7.0_09
(change the path to your JDK install path)
```

Also you must add your JAVA_HOME variable (add \bin\) to your path variable.

```
PATH =...;%JAVA_HOME%\bin\;
```

Maven

Download the current Maven from here and follow the instructions from the README: Extract it somewhere and create following path variables:

```
M2_HOME = C:\Program Files\apache-maven-3.0.4
(change the path to your maven folder)
M2 = %M2_HOME%\bin\
PATH =...;%M2%;
```

Eclipse

Download Eclipse from here. We recommend a current Eclipse version. We use Eclipse Juno (4.2) for Java developer. Remember to select the correct version for your operating system.

Maven Plugin for Eclipse

Open Eclipse and open "Help — Install New Software...". Now select or add the repository of your eclipse version. In our example we will use

`http://download.eclipse.org/releases/juno`

Select "m2e – Maven Integration for Eclipse" and click through the progress to install the plugin. If everything works well, Eclipse will ask you for an Eclipse restart.

Git or Egit for Eclipse

If you have installed Juno, Egit is already integrated. If not so, you can download Egit from here. Or you get and install Git from here.

Modelbus Team Provider for Eclipse

The update site for the Modelbus Team Provider is here. Select only the ModelBus Team Provider for the installation.

Modelbus and Metrino

Request the Modelbus developers for a version of Modelbus with an integrated Metrino service. Make sure that the following Jars lie in the plugin folder of Modelbus:

- de.fraunhofer.fokus.metrino.ruleEvaluator
- de.fraunhofer.fokus.metrino.measure

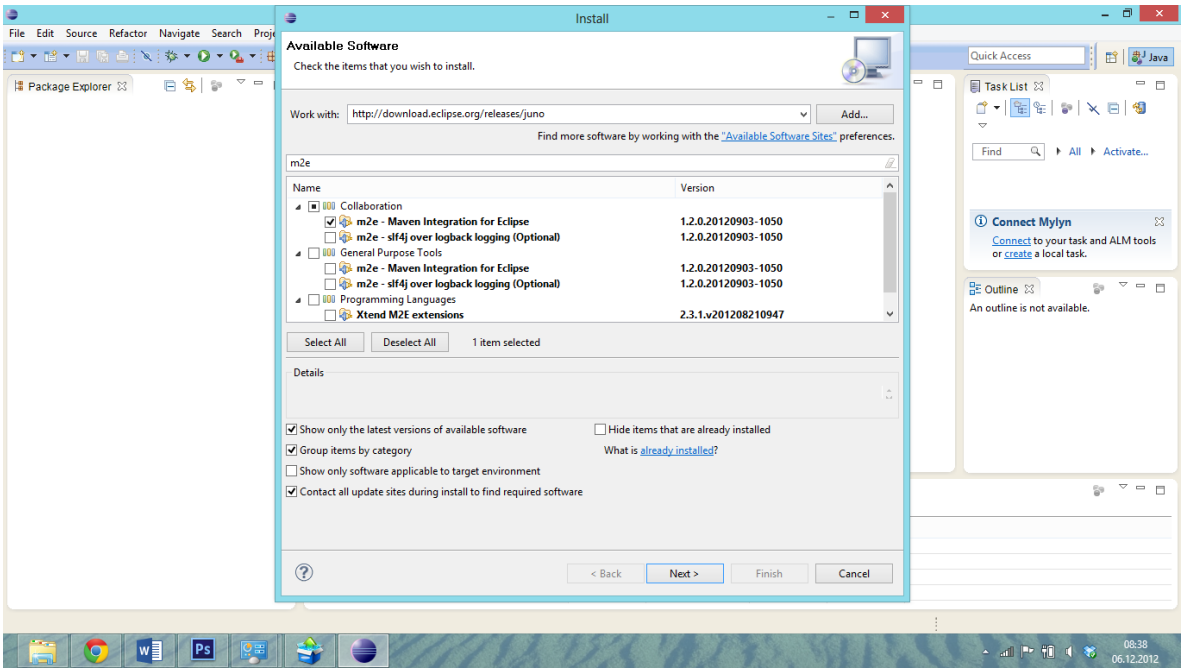

Figure 3.1: Maven Eclipse plugin installation

Extract Modelbus to any location in your file system. Now you have to setup the configuartaion file of Modelbus. Open the "modelbus.config" in the "serverConfiguration" folder of your modelbus installation. Set the "repositoryLocation" to "http://localhost:8080/modelbusrep and the "svnRepositoryLocation" to "repository":

```
<?xml version="1.0" encoding="UTF-8"?>
<config:ConfigModel xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
  xmlns:config="http://www.modelbus.org/system/model/config.ecore">
  <locations name="repositoryLocation" location="http://localhost:8080/modelbusrepository"/>
<!-- <locations name="secureRepositoryLocation" location="https://0.0.0.0:8181/modelbusrepository">
    <properties name="SSLTrustStore" value="SSL/cacerts.jks"/>
    <properties name="SSLTrustStorePassword" value="password"/>
    <properties name="SSLKeyStore" value="SSL/modelbus.keystore"/>
    <properties name="SSLKeyStorePassword" value="password"/>
    <properties name="SSLAlgorithm" value="RSA"/>
    <properties name="SSLPassword" value="password"/>
  </locations> //-->
  <locations name="notificationLocation" location="tcp://localhost:61616"/>
  <locations name="svnRepositoryLocation" location="repository"/>
</config:ConfigModel>
```

Then you will have to set some environment variables. Set

- MODELBUS_ROOT to the folder of your Modelbus installation

- MODELBUS_NOTIFICATION_LOCATION to "tcp://localhost:61616" like in the config file
- MODELBUS_REPOSITORY_LOCATION to "http://localhost:8080/modelbusrepository" like in the config file

3.3 Usage Manual

The typical workflow for plugin development is:

1. start Modelbus
2. open the plugin project with Eclipse
3. make some changes to the code
4. build the project with the Maven goal "install"
5. deploy Sonar including out plugin via Maven
6. analyse the project with the goal "sonar:sonar" to test the plugin on your installed Modelbus repository

The typical workflow for an user of our plugin is:

1. start ModelBus
2. start Sonar with our plugin via Maven
3. check in models into the Modelbus repository
4. analyse a Maven project with the goal "sonar:sonar" (this will also analyse your installed Modelbus repository)

The single steps are described below in detail.

Download the source of our Sonar-Modelbus-Plugin

Download the following copy of our repository:
`https://github.com/arsenij-solovjev/sonar-modelbus-plugin/archive/master.zip`

Extract the archive and if you like copy the folder sonar-modelbus-plugin-master into a special folder. It is just important that you don't copy this folder in your Eclipse workspace.

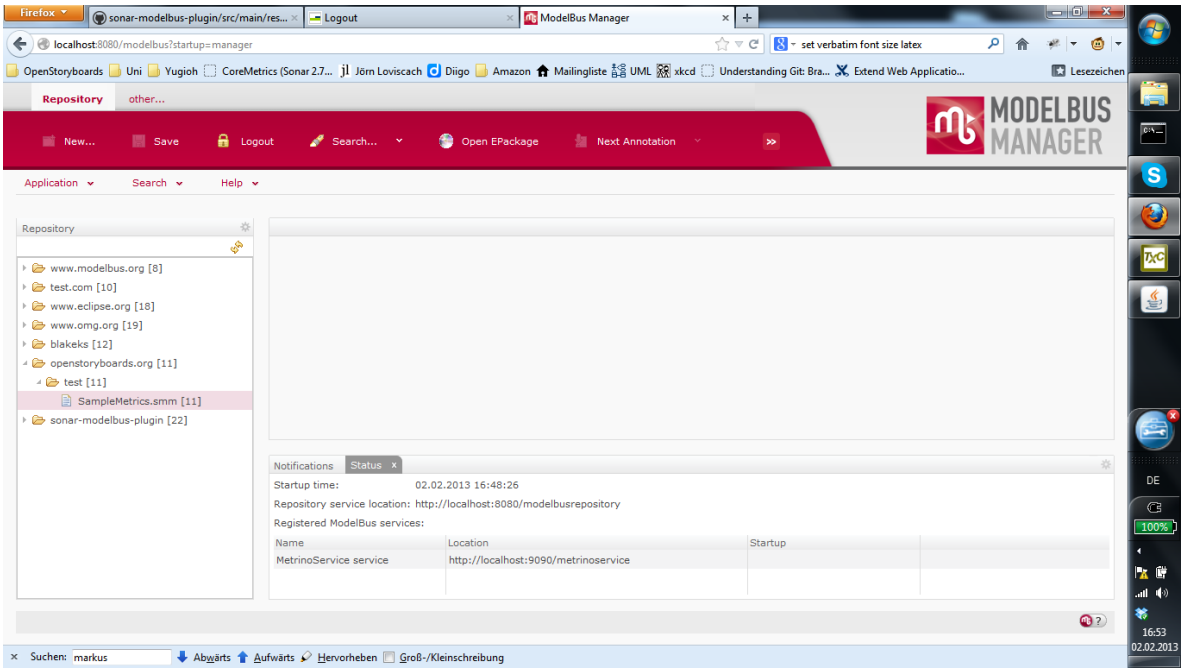

Figure 3.2: The Modelbus manager

Start Modelbus

Under Windows start the "startModelBusServer.exe" of your Modelbus installation.

Under Linux run the "startup.sh" as root user. Make sure that the "startup.sh" and the "bin/service" files are executable.

If Modelbus is up, you can visit the Modelbus manager under `http://localhost:8080/modelbus?startup=manager`: You can login with the username "Admin" and the password "ModelBus". Here you can see all the checked in files and models. When uploading a new file, press the refresh button to see the changes.

Uploading models

We created a tool for checking in and out files from/to the ModelBus repository. It can be simply called via "make". First compile the tool: Go to the "modelbusclient/sonar-modelbus-client" folder and run:

```
make install
```

This will compile and package the tool. Next, checkin some models to the Modelbus repository using:

```
make checkin
  URI=http://uri.de/location/in/repository/file.txt
  FILENAME=location/to/local/file.txt
```

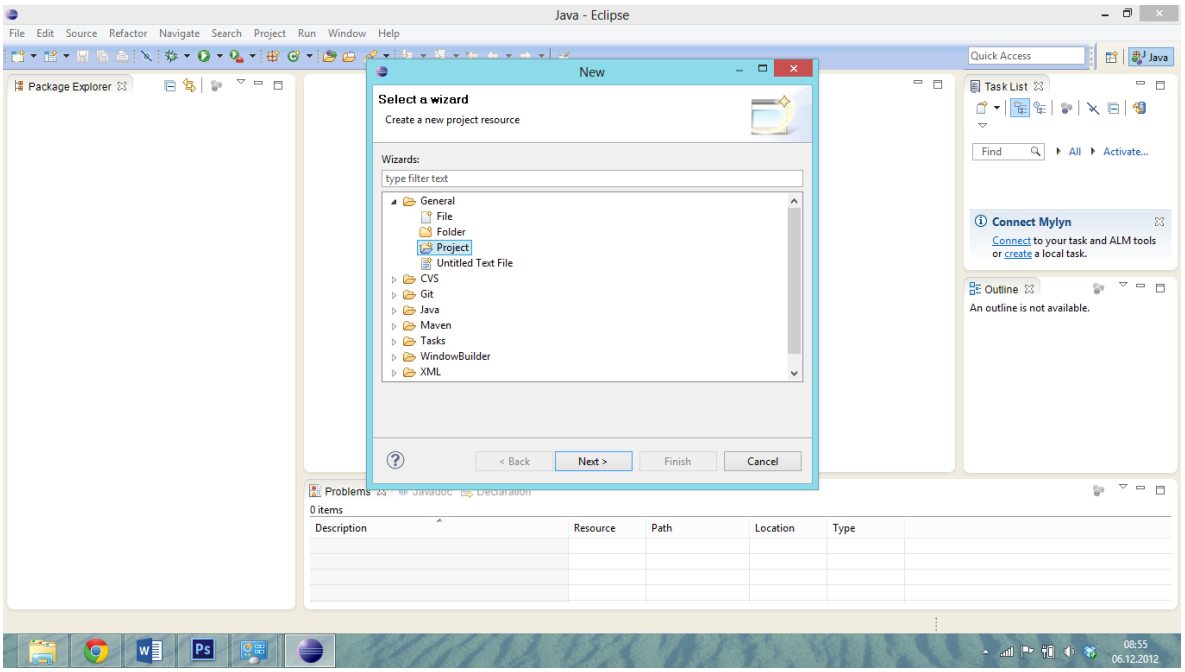

Figure 3.3: Creating a new Eclispe project

You can find some example models in the "/src/main/resources/metrinostuff" folder of our plugin.

Create a new Eclipse project

Now create a new Eclipse project:

Unselect the "Use default location" box and select the sonar-modelbus-plugin-master folder. Don't forget to set a project name.

Notice: If you finish the project creation process you may get "Git could not detect where GIT is installed" or "check HOME directory" warning but it doesn't matter in this case.

Convert your project to a Maven project

Right click on your project folder and choose "Configure — Convert to Maven Project".

Notice: If you get any errors (this is not unusual) just ignore them. If you can't do the next step just repeat the previous one.

Compilation

Next: right click on your project again and select "Run as — Maven install". If everything works well you see a "BUILD SUCCESS" reports like this:

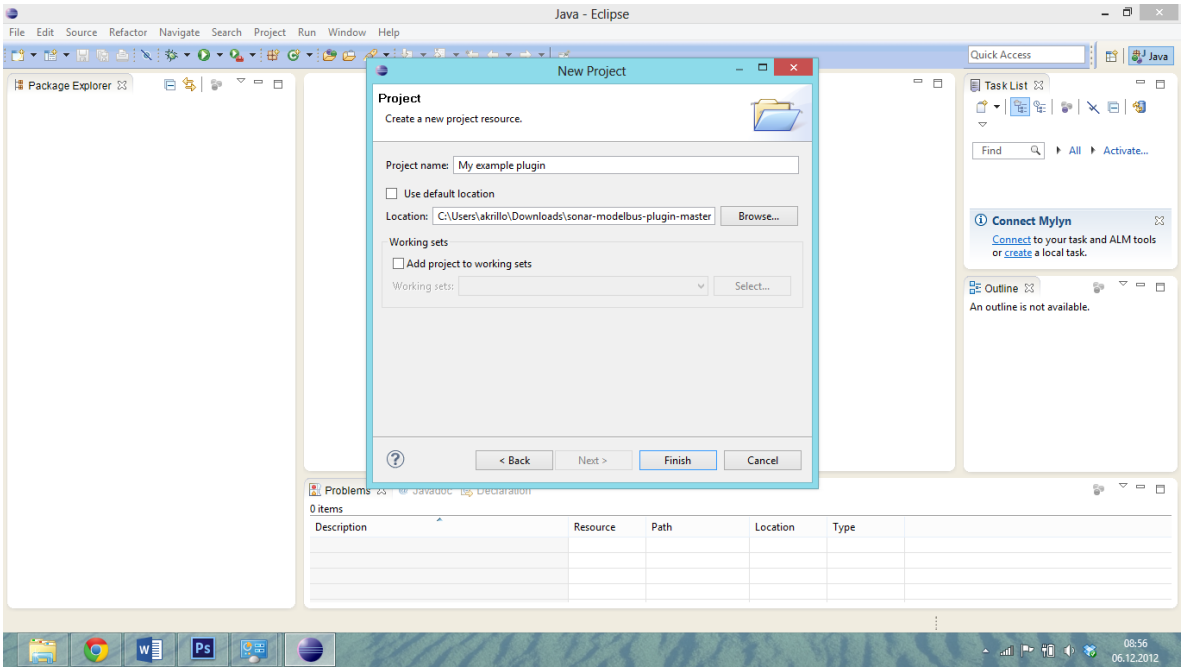

Figure 3.4: Selecting a location

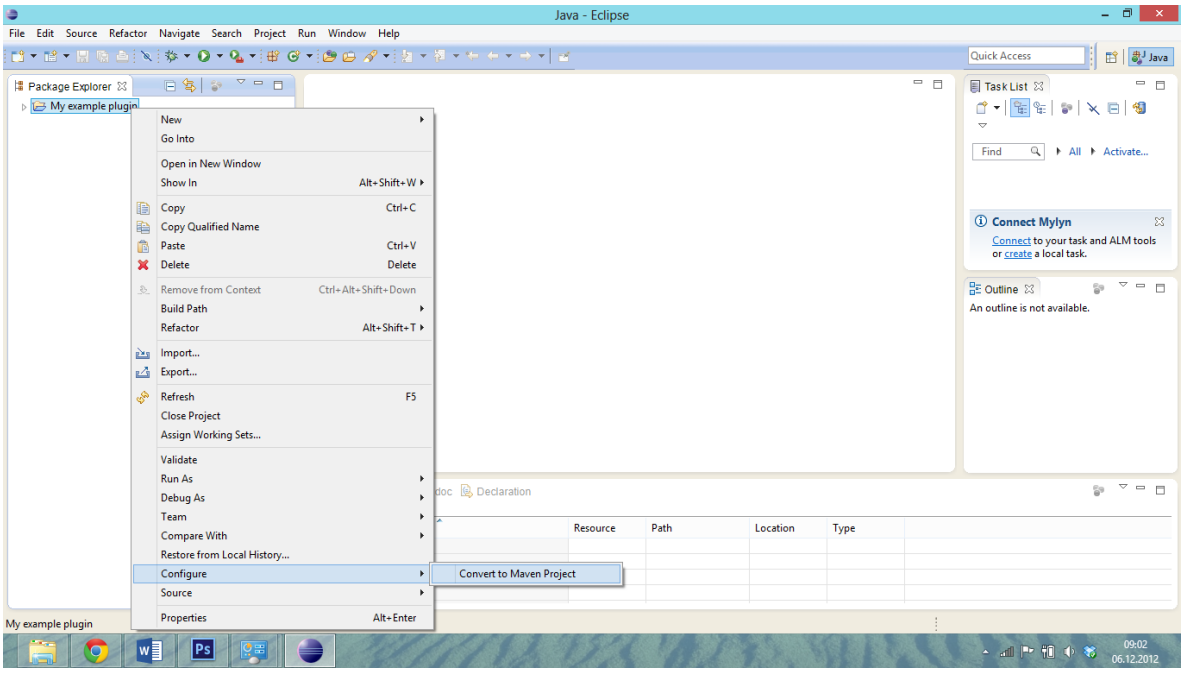

Figure 3.5: Converting to a Maven project

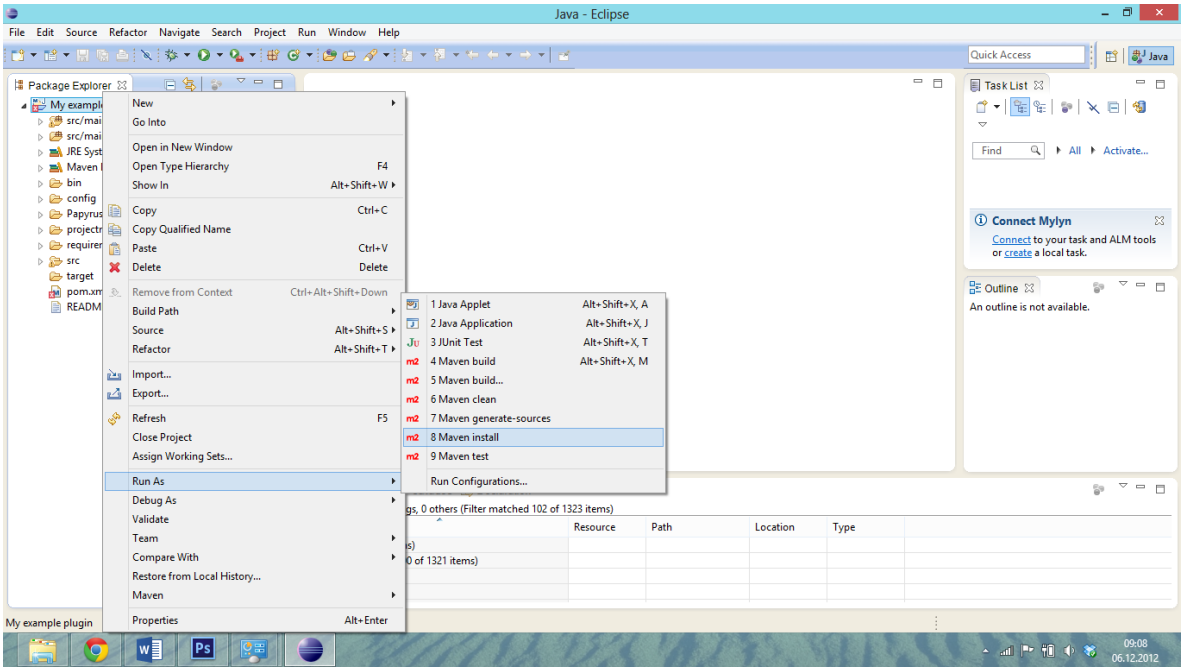

Figure 3.6: Maven install

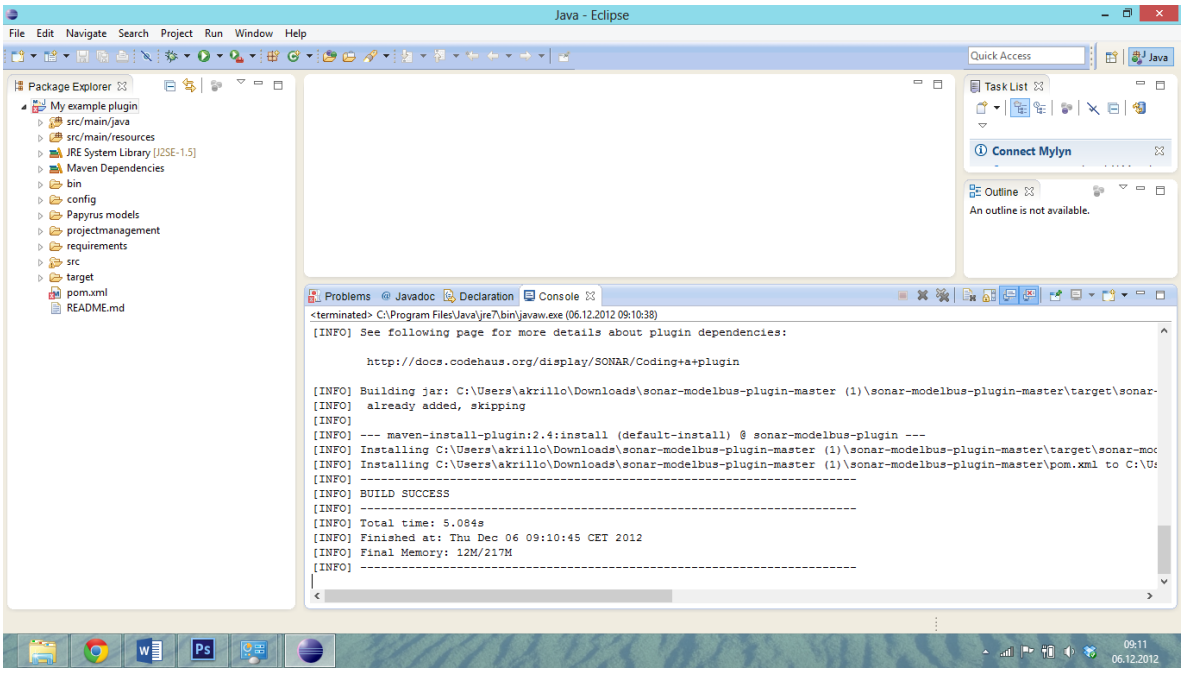

Figure 3.7: BUILD SUCCESS

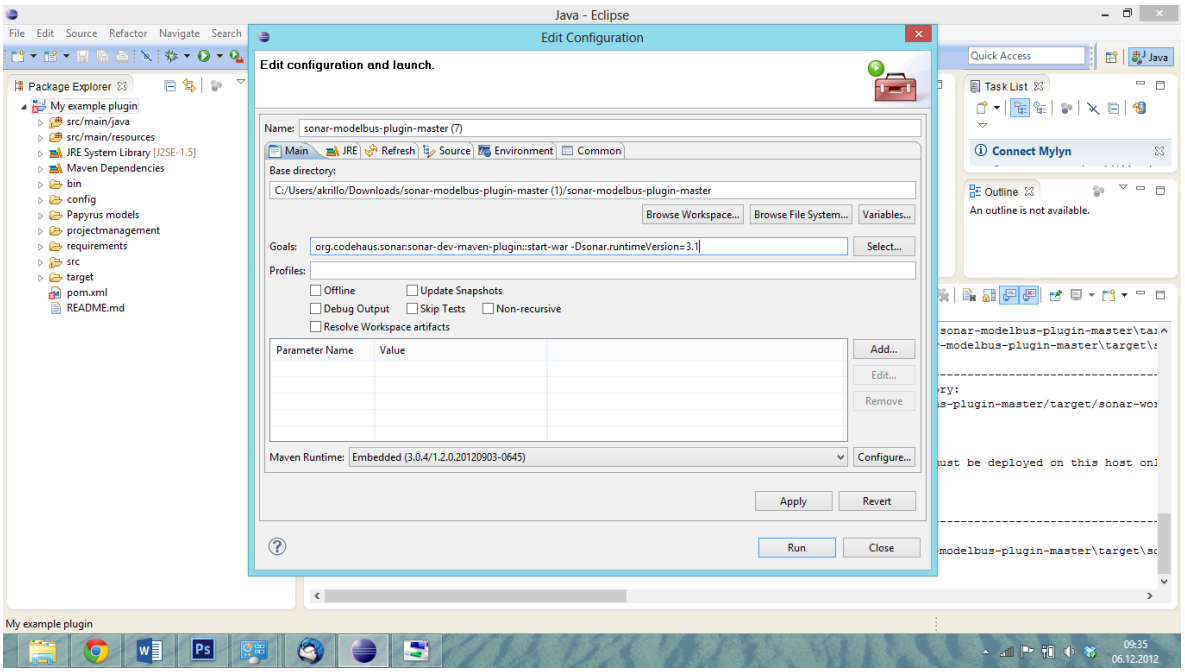

Figure 3.8: Deploying Sonar with Maven

Deploy the Sonar server

To run Sonar with our example plugin we need to run maven with an other goal. Right click on your Project and select "Run as — Maven Build". In "Goals" you must copy-paste

```
org.codehaus.sonar:sonar-dev-maven-plugin::start-war
-Dsonar.runtimeVersion=3.3
```

Then press "Run" to start the build process that will deploy a Sonar server for you.

Notice: this can take some minutes and if it seems to be frozen just restart the build process. If some errors occur just restart. If everything went fine you will see following status messages:

Now you can see Sonar running at "localhost:9000" in your browser.

Analysing with Sonar

If you like to analyse your project with Sonar (or any other Maven project) you can start the build process with the goal "sonar:sonar". After processing you can refresh your browser at "localhost:9000" and can see the results of the analysis.

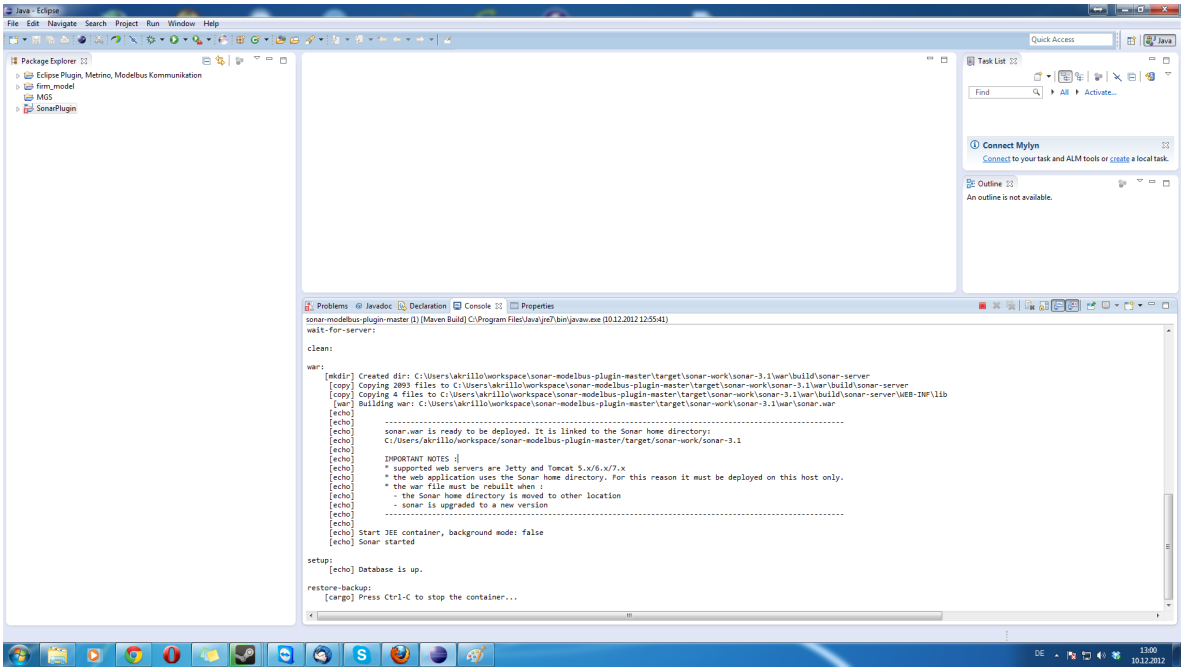

Figure 3.9: Sonar is ready

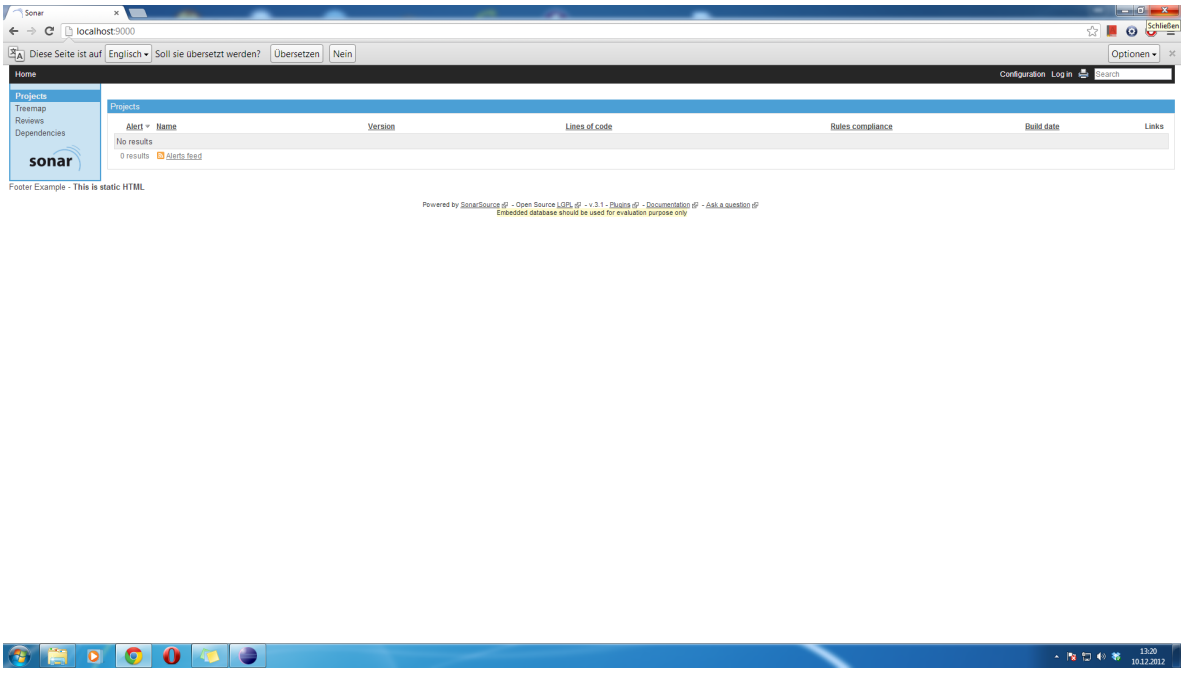

Figure 3.10: Sonar in the browser

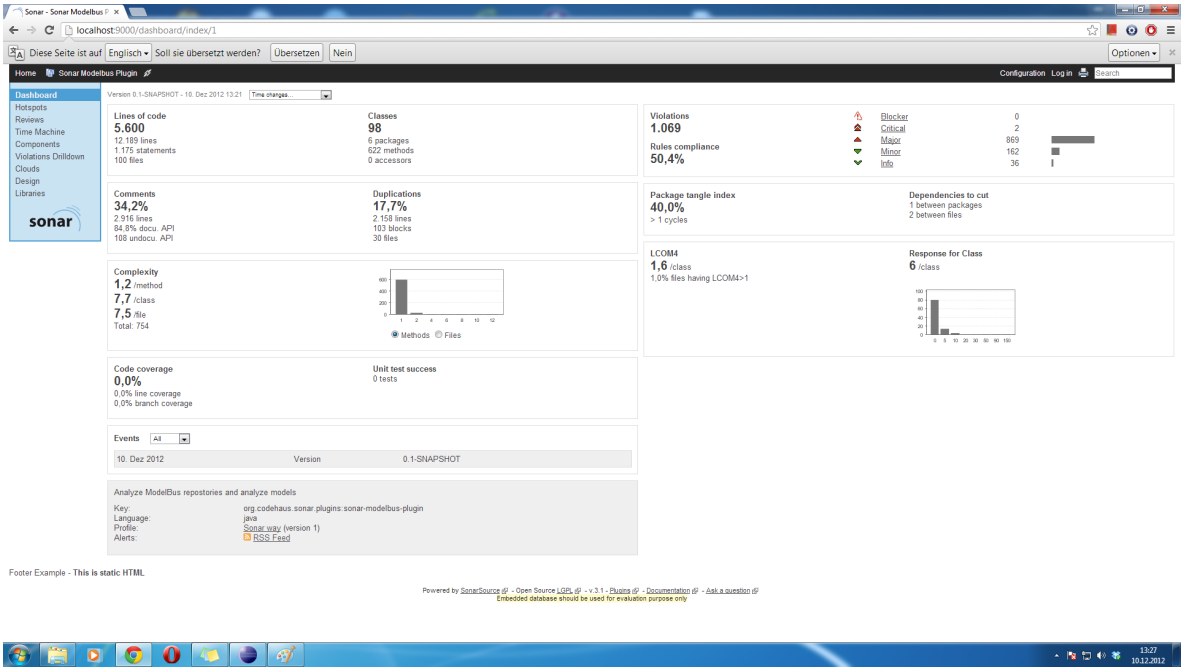

Figure 3.11: Sonar in action

Chapter 4

Conclusion

4.1 Review

Eigene Meinung. Was kann man besser machen im Kurs?

4.2 Conclusion

4.3 Closing Words

Dankeschön an alle Betreuer, wir haben viel gelernt.

Appendix A

Appendinx: Meetin protocols

A.1 Meeting 01 - 15.10.2012

The first meeting started with a presentation of the work of the *Frauenhofer Institute*, which is part of the *Frauenhofer Society*. The Frauenhofer Society is a German research organization with 60 institutes spread throughout Germany, each focusing on different fields of applied science (as opposed to the Max Planck Society, which works primarily on basic science).

The course is guided by Prof. Dr. Ina Schieferdecker. Our project group works with three computer scientists from the Frauenhofer Fokus institute in Berlin (Tom Ritter, Christian Hein and Michael Wagner).

At this meeting, we first got a general introduction to model driven engineering. Therefore we reviewed learning methods of abstraction to focus on creating and exploiting domain models (that is, abstract representations of the knowledge and activities that govern a particular application domain), rather than on the computing (or algorithmic) concepts. We also got a small overview over the main software development processes (waterfall model, v-model and so on). Our first task for the following week should be to create an UML diagram with Papyrus for a generic firm.

A.2 Meeting 02 - 22.10.20120

We presented all solutions for the task with the firm. There where surprisingly different solutions for the task. So there were actually no right or wrong solutions. For example: An employee was a full-time, half-time or even external worker. We modeled that with having an enumeration containing those items. Others used inheritance to model such a system. The next task was to create an UML profile for this firm.

A.3 Meeting 03 - 29.10.2012

The solutions of the UML profile were not as expected. All teams had different profiles and although there is no right and wrong again everybody had problems with Papyrus. In many cases Eclipse just did not react to changes in the diagram or did not do the expected operations. Only one person had no problems with Eclipse, so this person presented its profile. So the difficulty was not to create the profile, but to get this plugin working.

A.4 Meeting 04 - 05.11.2012

The first meetings were like an introduction to model based engineering. We got a few tasks to do. In this meeting the project tasks were presented. There were a bunch of tasks. In most cases a tool for a special case was needed and so we should implement it. We wanted to take the Sonar task and we were lucky, because nobody else wanted to take this task. With the teams we made a small brainstorming to get a feeling for the task. In this brainstorming we collected our knowledge about static code analysis and especially Sonar. Because of the fact some team members already had worked with Sonar, they gave us a short introduction to code analysis with Sonar.

The task for the following week was to install and test Sonar and the Modelbus repository, which is a system containing models and source code, that should be analyzed with Sonar (that was our task). Another task was to work out some first requirements.

A.5 Meeting 05 - 12.11.2012

We presented the requirement list, which of course was not specific enough. We simply did not have much knowledge about the task and about plugin development for Sonar. Other teams had the same issue, their results were not specific as well.

The task for the following meeting was to specify the requirements more and to provide a first architecture concept. You can find the new requirement list here.

A.6 Meeting 06 (Hackathon) - 23.11.2012

The weekly meeting at university was not taking place this week, so we decided to meet and work together in the team. We therefore organized a hackathon, where we installed the software (some had build problems during the installation of the components). We also worked out an architecture concept and created a few diagrams to show our progress.

A.7 Meeting 07 - 26.11.2012

We presented our new requirements and the architecture. Our architecture concept (Figure A.1) and the workflow (Figure A.2) was regarded to be good enough, so we are good to start the development of the plugin. From now on, we focus on more technical tasks.

A.8 Meeting 08 - 03.12.2012

At this meeting, we presented the current results. The result is, that our WSDL client can connect to the Modelbus repository. The Modelbus service is now successfully

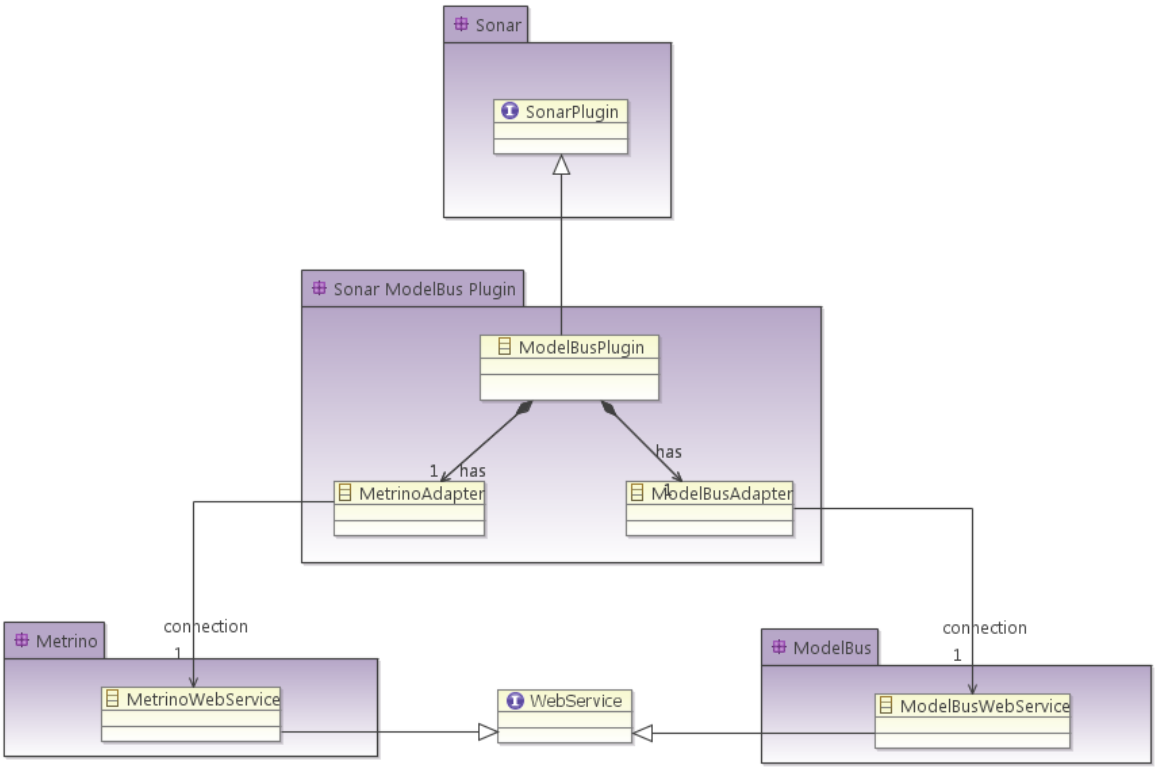

Figure A.1: Architecture of the Sonar plugin

driven on our test system and it is planned to connect successfully to the Metrino service.

In the discussion with the project head members, they suggested to use a customer instead of a WSDL client to connect to the Modelbus repository. The customer is a type of client service, which do not communicate with the service with WSDL but instead connects directly to the Java interface of Modelbus.

After discussing this point, other questions raised concerning object-oriented metrics for models:

- What kind of properties can be analysed on object-oriented code?
- Into what extend can analyse of object-oriented code applied to object-oriented models?
- Requirement: Sonar shall be capable to analyse object-oriented code, too. These analysis shall be applied to models, too.

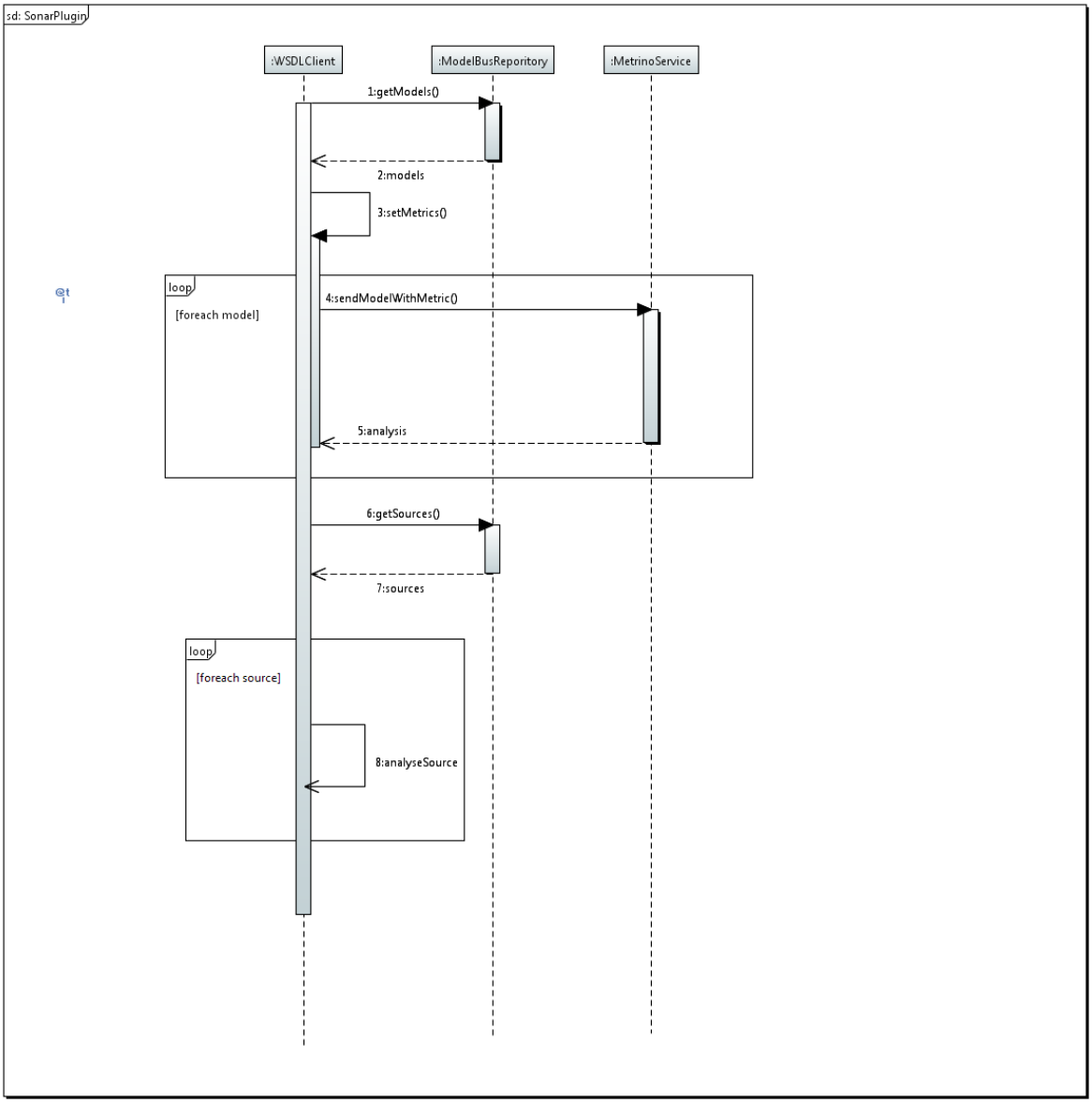

Figure A.2: Sequence diagram, showing the activity of our plugin

A.9 Meeting 09 - 10.12.2012

At the beginning of this meeting, we presented our current results and reported some problems. Our biggest problem was, that during the installation process of Metrino, a lot of Eclipse libraries must be installed by hand. As we reported this problem to the organizer of this course, he answered that in theory, a manually installation of Eclipse libraries is not required but in practice, some Eclipse libraries must be integrated.

Another discussion point was, that we wrote a message to the developers of Modelbus and Metrino and asked them about a consumer, which can connect to the Modelbus repository without an own WSDL client. We tried it with the consumer but unfortunately, it did not work. Therefore, we connected manually to the Modelbus repository. Additionally, we wanted to connect to the Metrino service but we discovered, that Metrino can just be reached locally. Even in the same network, the connection did not work.

We explained our organizer, that we still not know how to send our metrics to Metrino, thus we need an API documentation.

We told our organizer about the complex installation process of Modelbus: To configure Modelbus, the Modelbus-Root for the local service must be pointed to a config file, which points to a remote server.

An additional discussion point was, that there is no list of dependent libraries for the configuration of Modelbus.

Our organizer told us, that he will send us a front-end for the Metrino service, so that we can understand which metrics Metrino can use.

The last point of discussion was, that one of our team members had the idea, that models can be translated into Java code, which can then be analyzed statically by Sonar. But this would break a requirement, which says that a model must be analyzable, too.

A.10 Meeting 10 - 17.12.2012

During the last week, we discovered the problem that the checkout of the models from the Modelbus repository did not work. We always got an exception `ServiceConstructionException`. In the standalone version of our plugin, the checkout worked. We explained this error to our supervisor and he explained that we must send this problem to their mailing list. Then, we asked them a few questions about some functions of the Metrino service, which occurred during the work with it.

A.11 Meeting 11 - 07.01.2013

We reported the current state of our plugin project. Now, the Metrino service runs on our own server on *openstoryboard.org*. The problem is, that we want to reach the service from the outside of the network. Therefore we need a proxy.

We checked in some models into the Modelbus repository. We explained that we still get some exceptions during the checkout of some models from the repository. Because

of that, we asked for the source code of Modelbus so that we can debug the code. The supervisor suggested, that one member of our team could visit him the next day at Frauenhofer so that they can debug our code together.

Additionally, we explained that Sonar is not designed to analyze code in different languages within one process. To be able to analyze different kind of languages, a separate module must be run for each language and therefore, cannot be done within one plugin. To analyze different languages, we must write a plugin for each language which is not the object of our plugin project.

A.12 Meeting 12 - 14.01.2013

To summarize our work, we explained how our plugin works and for what it is designed for. We explained, that we checkout models from the Modelbus repository, that we send them with metric descriptions to the Metrino service and get the result file and finally, that we already have a module, which can visualize results in Sonar.

The next task is to build a SMM Parser which can read the result SMM file, which we get from the Metrino service, and to store the results in data structures, which are read by Sonar.

The example SMM files, which we got from our supervisor, computed two metrics: the number of classes and the number of packaged of a metric. We have seen that there are more than just one result value for a metric computation. Therefore, we must know which one of them is the newest one. Our supervisor explained, that the XSD of the SMM file explains that each result can have a time stamp so that we can see easily, which one is the newest metric result.

A.13 Meeting 13 - 21.01.2013

This week, the meeting did not take place. Our group organized a Skype meeting where we discussed the results of our tasks and what are the tasks for the following week. The visualization of results in Sonar still not works properly and we must go on with these tasks. Additionally, we updated our requirements list. We discussed about the SMM parser, which was built until this day and realized, that there are some errors which must be fixed this week, because next week, we want to put all modules of the Sonar plugin together.

A.14 Meeting 14 (Hackathon) - 26.01.2013

Today, we have a hackathon. We have a lot of results for each little module of our plugin project and we want to put them all together.

At first, we integrated our SMM parser into our plugin. We changed the parser a little bit. We added an input stream to the Resource object, because by doing this

we can send the checked out models directly from the main application to our parser instead of saving the checked out models in a file.

Afterwards, we work on metrics for Metrino. We have got two example metrics, but we want to define some more metrics. We have written them in OCL. For a few weeks, our supervisor sent us a Modelbus eclipse project where they can create metrics within a GUI perspective. In some cases, we got an exception so that we started to write metrics in OCL by hand.

Finally, we discussed about the next tasks and steps.

A.15 Meeting 15 - 28.01.2013

We reported the current state of our project. There are still two requirements, which are not satisfied. One metric is, that we analyze more than just one language within our plugin, but in discussion with the developers of Sonar, we got to know, that this is not possible within a single Sonar plugin. The second requirement is the question, how many other metrics we shall create to analyze models.

We thought about two ways: One way is to write a lot of metrics by hand and statically integrate them in our plugin. Thus, if a user wants to have additional metrics, he or she must insert them manually. Another way is to write an extension for our plugin so that the user can write some metrics and our plugin can load them dynamically. In general, Sonar plugin uses metrics which are added manually. This means that if we want our plugin to be statically load the metrics, we must write them all by hand into our classes, thus we must know all metrics a user possibly want to analyze in Sonar or the user have in the SMM files. We decided to write a mechanism to load dynamically metrics.

Our supervisor recommended us to take all metrics, which Sonar uses to analyze source code in Java. Therefore, we must examine which of the metrics in Sonar we can adapt to analyze models.

A.16 Meeting 16 - 4.02.2013

The last week, we tried to find out how we can dynamically load metrics into our plugin, so that a user do not have to write them by hand. We explained how much we could find out and additionally, we presented the first version of our documentation and got some constructive criticism.

A.17 Meeting 17 - 11.02.2013

In the last meeting, we presented the current state and which sub-tasks are left to finish the plugin development. We explained that we still have problems to visualize the results of Metrino in Sonar and which exceptions we got. Our supervisor told us to meet him after the lecture so that we can take a look at the problem. Finally,

we presented him the current version of our documentation. There are some good points, for example that the software architecture should be the main part in our documentation (but we placed it after the installation manual).