HSR – University of Applied Sciences Rapperswil

Institute for Software

Semester Thesis

metriculator
CDT metric Plug-in

Ueli Kunz, ukunz@hsr.ch
Julius Weder, jweder@hsr.ch

http://sinv-56013.edu.hsr.ch

Supervised by Prof. Peter Sommerlad

December 22, 2011

Abstract

This thesis aims at statically analysing software written in C++ using known software metrics. Software metrics are used to obtain objective, reproducible and quantifiable measurements of source code. This measurements may support various tasks such as performance optimization, quality assurance testing or software debugging [wik11].
Metriculator is programmed in Java and integrates in Eclipse as plug-in that depends on the Codan [cod11] framework which is part of the C/C++ Development Tooling platform (CDT, [CDT11]). Codan is a code analysis framework that offers a mechanism to add new code analysis features. Each metric in metriculator is implemented as an extension to Codan. First and foremost it is about extracting meaningful information out of C++ source code using different software metrics.

After analysing the code, the results are shown in the metriculator view, which provides different representations of the same underlying data. Additionally the metriculator view helps finding problems and moving to their problematic source code sections. Optionally the detected problems will be displayed as markers with detailed problem information within the source code editors. Each metric can be activated or deactivated and as well have variable threshold values that suite the needs of the specific domain.

There are five metrics already implemented and it is possible to extend metriculator with additional metrics without touching the existing source code. Implemented metrics:

- Number of Logical Source Lines of Code (LSLOC)
- Cyclomatic Complexity (McCabe)
- Number of Parameters per Function
- Number of Members per Type
- Efferent Coupling per Type

Management Summary

This chapter summarises the goals and outcomes of this thesis. And gives a preview on what might be possible in the future.

Initial Situation

Various aspects of source code can be expressed in numbers. Such an aspect is called a software metric. For instance, a well known software metric measures the number of lines of code. Software metrics allow users to view the analysed source code from different perspectives. They may give a high level overview of the size and quality of the analysed source code. But they can also help developers to identify problematic source code sections.
Static analysis tools investigate source code without running the application itself. Many static analysis tools exist for various programming languages [met11b]. But at the time of this project, there was no metric tool available that integrates into the Eclipse C/C++ Development Tooling platform (CDT, [CDT11]). Our goal was to create the first static source code analysis tool for CDT.

Procedure and Technologies

metriculator is a plug-in for CDT that itself is a plug-in of the Eclipse framework. CDT is a well known platform to develop C/C++ software. Beside CDT, metriculator uses a framework called Codan [cod11] to analyse C++ source code. Codan is extensible so that third party tools such as metriculator can hook in and add their own code analysis features.
Metriculator analyses a user defined set of files. The analysis results are applied to functions, classes, namespaces, files, folders and projects. The analysis results are viewable using different views that all rely on the same underlying data.

Results

Metriculator supports the following five software metrics, each of which is described in detail in section 3.

Logical Source Lines of Code numbers the size of the software by counting the lines of code. See 3.1 for detailed description.

Cyclomatic Complexity also called McCabe, is an indicator for the complexity of the software. For instance many *if* statements increase the complexity and lead to unreadable and hard to maintain code. See 3.2 for detailed description.

Number of Parameters per Function measures the number of parameters of a function. See 3.3 for detailed description.

Number of Members per Type measures the number of members of a class, struct or union. See 3.4 for detailed description.

Efferent Coupling counts the number of foreign types a certain type depends on. For instance, many dependencies indicate highly coupled software. See 3.5 for detailed description.

Compared to the specified objectives at the start of the project there are two metrics missing. The Number of Nested Levels and the Number of Template Parameters metric were not implemented. This is because we decided to implement the Efferent Coupling metric in advance and because of the unforeseen performance issues that must have been resolved prior to additional feature implementations. See 5.1 for more details.

Metriculator is available as Eclipse plug-in and can be installed from within Eclipse using the install wizard. After installation, users can run the Codan code analysis. As part of this analysis, metriculator will run as well. The gathered metric data can be investigated using different views provided by metriculator.

Illustrated Example

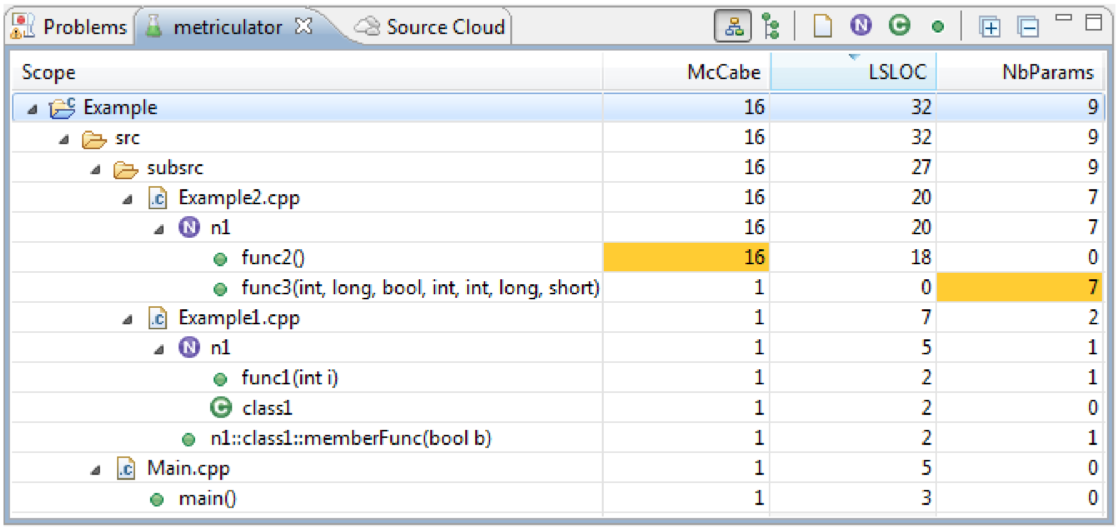
In this section we will explain the results, shown in the metriculator views, after analysing a sample project.

Default View Figure 0.1 shows a screenshot of the default metriculator view after the static code analysis ran. The default view shows all analysed files and folders in a tree view. Files can further be expanded to explore the source code they contain. The values of each metric are displayed in a separate sortable column. Each row represents one analysed scope.

Warnings As visible in Figure 0.1 *func3* has seven parameters. Because seven is above the threshold set in the metric preferences the cell is highlighted.

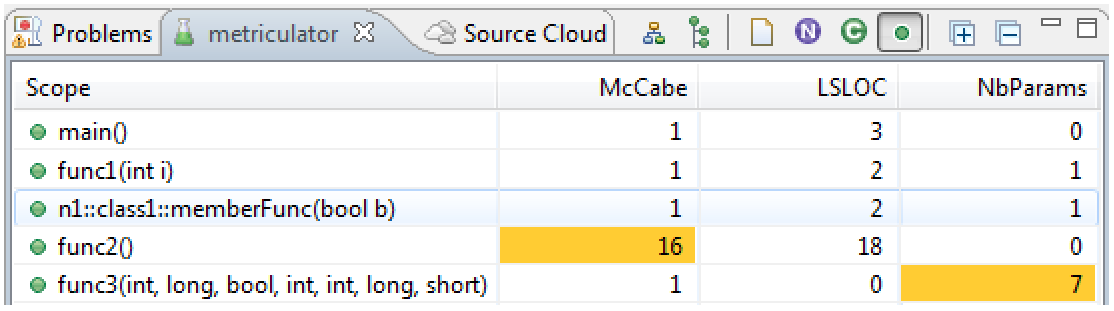
Filter View The filter view in Figure 0.2 shows all functions. Other filters can be applied to see a list of only files, types or namespaces. Thanks to the filter view, users are able to quickly identify large files for instance.

Data Visualisation Metric values can also be visualised in a tag cloud, as illustrated in Figure 0.3 using the LSLOC metric values. The bigger the font size of the function name, the higher its metric value is.



Scope	McCabe	LSLOC	NbParams
Example	16	32	9
src	16	32	9
subsrc	16	27	9
Example2.cpp	16	20	7
n1	16	20	7
func2()	16	18	0
func3(int, long, bool, int, int, long, short)	1	0	7
Example1.cpp	1	7	2
n1	1	5	1
func1(int i)	1	2	1
class1	1	2	0
n1::class1::memberFunc(bool b)	1	2	1
Main.cpp	1	5	0
main()	1	3	0

Figure 0.1.: Screenshot of the metriculator hybrid view.



Scope	McCabe	LSLOC	NbParams
main()	1	3	0
func1(int i)	1	2	1
n1::class1::memberFunc(bool b)	1	2	1
func2()	16	18	0
func3(int, long, bool, int, int, long, short)	1	0	7

Figure 0.2.: Screenshot of the metriculator filter view listing all functions.

Future Work

There are many interesting not yet implemented metrics that would increase the value of metriculator, see chapter 1.2.1 for an incomplete list.
Performance can further be improved to allow analysing source code with over about 300'000 physical lines of source code. Fixing that issue would make metriculator even more attractive to analyse large projects. Other unresolved issues are listed in chapter 5.2.
A reasonable application of metriculator in the future is to serve as refactoring assistant. Based on a reported problem, metriculator could suggest a refactoring or quick fix that solves that problem.

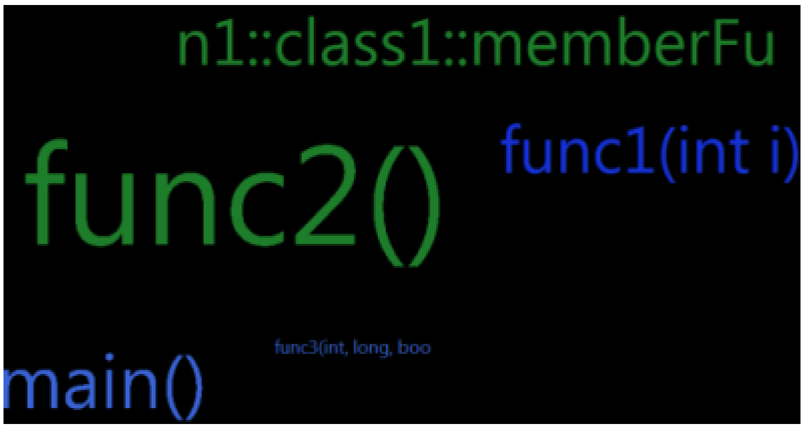


Figure 0.3.: Generated tag cloud that visualises the LSLOC value of all functions.

Contents

Experience is a hard teacher because she gives the test first, the lesson after ward.

Vernon Law

1. Introduction

The investigation of source code has ever since been an important task in software development [his11]. Static source code analysis is one type of source code investigation with the objective to get a picture of the written code itself. One output of static source code analysis are source code metrics. Many types of metrics exist since source code can be inspected under different aspects. For instance the lines of code metric gives an impression about the size of the code.
Many metric analysis tools for almost any programming language already exist. But at the time of this semester thesis there was no official C++ metric tool available that integrates well in to the Eclipse CDT (C/C++ Development Tooling) platform [CDT11].

1.1. Motivation

We intend to improve the quality of the Eclipse CDT platform as it will help other developers to create better software. Code metrics provide a big picture of the quality of the source code. There are plenty of well known and also standardised code measurement techniques around and implemented for other tools and languages. We give our best to implement these techniques and standards for the CDT platform.

1.2. Objectives

Project organization Fixed one-week iterations are used. Redmine [red11] is used for planning, time tracking, issue tracking and as information radiator for the supervisor. A project documentation is written. Organization and results are reviewed weekly together with the supervisor.

Integration and Automation Sitting in front of a fresh Eclipse CDT installation a first semester student can install our metric plug-in using an update site as long as metriculator is not integrated into the main CDT plug-in. An update site is created to allow the installation of metriculator using the Eclipse install wizard.

Quality The plug-in code is covered with automated test cases. Automated UI tests are not mandatory.

Delivered Assets At the end, the project will be handed to the supervisor with two CDs and two paper versions of the documentation. The CDs contain: this project report, a video demonstrating the usage of metriculator, the source code and the deployable plug-in.

Implemented metrics The metrics listed below are going to implemented with highest priority.

- Number of Members per Class
- Lines of Code per File
- Lines of Code per Class
- Lines of Code per Function
- Cyclomatic Complexity (McCabe) per Function
- Number of Nested Levels
- Number of Template Parameters
- Number of Parameters per Function

1.2.1. Advanced Objectives

If the basic objectives get finished before the end of the project, we may start implementing the following metrics:

- Number of References of Type (class, struct, union)
- E/Afferent Coupling
- Feature Envy
- Lack Of Cohesion per Function
- Number of Overloads
- Depth of Inheritance Tree
- Instability
- Abstractness
- Normalized Distance from Main Sequence
- Number of Locals in Scope
- Number of Lines of Code vs Lines of Comment (Density of Comments)

1.2.1.1. Tag Cloud Integration

Tag clouds are an excellent way to visualise weighted data. The Sourcecloud plug-in for Eclipse [sou11] generates a tag cloud of all words in source code files. The view of that plug-in could be integrated in our plug-in to visualise weighted data of one metric. For instance, the LSLOC values of all functions. A higher LSLOC value results in a bigger font size of the function name.

1.3. Project Duration

The semester thesis starts on September 19th and has to be finished until December 23rd, 2011.

2. Requirements

This chapter describes the functional and non functional requirements of the metriculator plug-in.

2.1. metriculator view

This view is able to show the data of all metrics for different scopes. It visualises the underlying data in three different views.

hybrid view is the default view and shows the projects and its content as a mix of physical and logical representation in a tree. This is the default view and shows a tree view representing the file system structure of the investigated files and folders, where the files further contain the logical tree structure that is a simplified form of the abstract syntax tree (AST) of the file contents. The following node types are displayed:

- project
- folder
- file
- namespace
- type (class, struct, union)
- function

logical view shows the logical representation of the source code in one tree. This view merges the logical trees of all files and thus does not show any file system information. The following node types are displayed:

- namespace
- type (class, struct, union)
- function

filter view shows a flat representation of all hybrid tree nodes. The node list can be filtered by the different types of nodes. Following filters exist:

- file
- namespace
- type (class / struct / union)

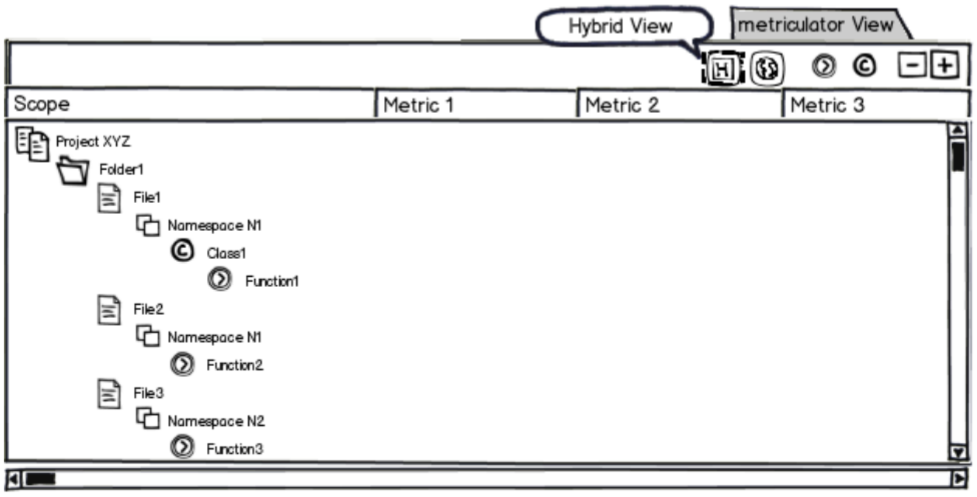


Figure 2.1.: Hybrid view

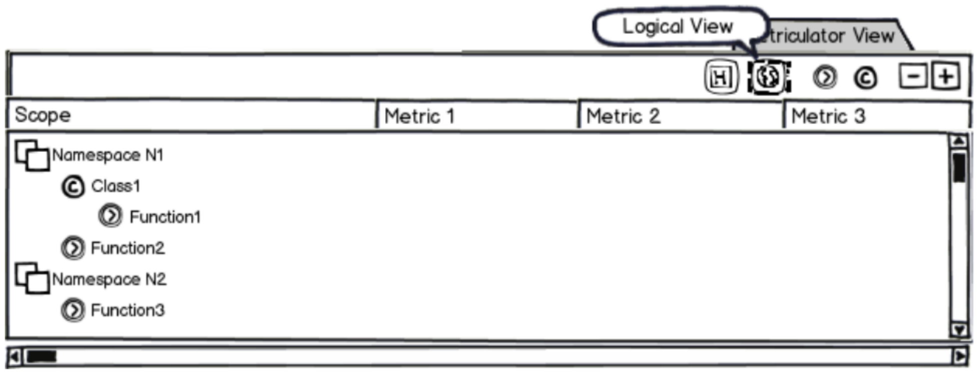


Figure 2.2.: Logical view

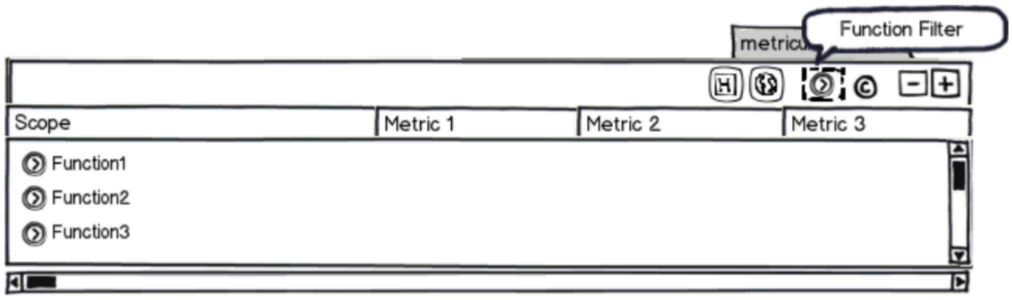


Figure 2.3.: Filter view

- function

tag cloud view is another nice way of visualising metric values by generating a tag cloud that allows fast visual detection of potentially problematic items such as large functions or high coupled classes (Figure 2.4).

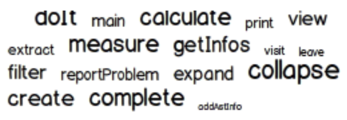


Figure 2.4.: Tag cloud view

In all views the columns can be sorted ascending or descending, by scope name or metric value. The nodes in the tree structure can be expanded and collapsed (fully or partially).

2.2. Use Cases (Brief Format)

The following use cases apply to all implemented metrics.

2.2.1. Starting the Metrics Analysis

A user chooses some C++ projects he wants to analyse and runs the Codan command in Eclipse. All calculated metrics will be displayed in the metriculator view.

2.2.2. Change Metric Configuration

A user can change the configuration of the Lines of Code metric by specifying an individual range of acceptable thresholds. The view marks the values outside of the thresholds.

2.2.3. Visualise Metric as tag cloud

The user chooses a metric in the metriculator view. Metriculator generates a tag cloud which visualises the values of the selected metric.

2.2.4. Analyse Metric Results

The user has run the analysis and the metriculator view is shown. He can now switch between the different views, sort columns, or generate a tag cloud.

3. Metric Specification

This chapter specifies the details of the implemented metrics. Most metric implementations follow standards (specified in sub chapters) with some extensions where required by C++. Each metric specification describes what the metric means, which scopes it applies to and how it is measured.

3.1. Logical Source Lines of Code (LSLOC)

Source Lines Of Code (SLOC, or Lines Of Code) [lsl11] is a unit used to measure the size of software programs. There exist different types of Lines Of Code (LOC) metric definitions. Some count the number of statements, others count the physical lines of source code (with or without comments) or even the number of byte code instructions. We decided to use the term LSLOC (Logical Source Lines of Code) because the University of Southern Carolina defines a standard for LSLOC counting rules for C++ [lsl11]. As long as not otherwise noted metriculator respects the rules of this standard. Because this standard does not define rules for some of the new concepts of the C++11 [cpp11] standard, especially lambdas, we defined new rules where required.

3.1.1. Lambda Expressions

```
int z, x;
auto square =
    [z, x]        // 1 - one lambda-capture, two captures
      ()
        {       // 1 - function body
        return z*z; // 1 - statement
        }
        ;       // 1 - assignment statement
```

Listing 3.1: Lambda expression code snippet with LSLOC of 4

As illustrated in listing 3.1, the following rules apply to lambda expressions:

1. A lambda-capture counts one, regardless of the number of captures it contains.
2. Anonymous function declaration counts one (as any other function declaration does).
3. Statements in the lambda body count as usual.

4. The assignment of the lambda expression to *square* counts one (as any other assignment).

5. Direct calls of lambda expressions (no assignment) count one.

In contrast to other function declarations, lambdas may have a lambda-capture that increases the LSLOC value by one.

3.1.2. Enum

```
enum x{
  val1,
  val2
} // 1 - type definition
```

Listing 3.2: Enum code snippet with LSLOC of 1

As illustrated in listing 3.2, the following rules apply to enums:

1. An enum declaration counts one.

2. Enum values do not count.

3.2. McCabe (Cyclomatic Complexity, CC)

The McCabe metric, also known as Cyclomatic Complexity [mcc11], is a software metric for measuring the complexity of parts of a software. This metric indicates whether a piece of code is still comprehensible to humans. Simply expressed it is the number of binary branches plus one.
Our implementation is based on a paper published by Verifysoft Technology GmbH [mcc11].

3.2.1. Example

```
namespace n1 { // 1 + 2 + 1 - 2 = initial + func1 + func2 - #children

  int func1(int i){ // 1 + 1
    if(i < 0){ // 1
      return 0;
    }
    return 0;
  }

  int func2(int){ // 1
    return 1;
  }
}
```

Listing 3.3: Code snippet example McCabe

Every node has an initial Cyclomatic Complexity of 1. In listing 3.3 the function *func1* has due to the *if* statement a Cyclomatic Complexity of 2. The parent node of the two functions has therefore a Cyclomatic Complexity of 2. To build the node value of a parent node, all children node values are summed up. Afterwards the sum of the parent is decremented by the number of children.

3.3. Number of Parameters per Function

The Number of Parameters per Function metric is a software metric, that counts the amount of parameters in a function or member function. All parameters are counted. This metric is an indicator whether a function is painful to call or maybe degrade performance.

3.3.1. Exceptions

1. If a function definition has a forward declaration, the parameters of the forward declaration are not counted. Following Listing 3.4 illustrates an example:

```
int doIt(int, bool);  // does not count because of the definition below
int doAnother(int, bool); // counts 2 parameters

int doIt(int i, bool b){  // counts 2 parameters
  return b ? i : 0;
}
```

Listing 3.4: Code snippet for Number of Parameters per Function - function declarations and definitions

2. If a member function declaration has an associated definition, the parameters are not counted in the logical view but they are counted in the hybrid view, expect they are in the same file, then the hybrid view also does not count the function declaration. Following Listing 3.5 illustrates an example:

```
class Example{
public:
  void doIt(int, bool); // does not count because of the definition below
  void doAnother(int, bool);  //counts 2 parameters
};

void Example::doIt(int i, bool b){  //counts 2 parameters
  // do something
```

```
    }
```

Listing 3.5: Code snippet for Number of Parameters per Function - member function declarations and definitions

3.4. Number of Members per Type

The Number of Members per Type metric is a software metric that counts the members of a type. A types can be a *class*, *struct* or *union*.
Our implementation is based on the official C++11 Standard [cpp11] especially on the chapter Class Members. As long as not otherwise noted, metriculator respects the rules in this standard.

3.4.1. Explanation of Member Types

There are three different kinds of members:

1. data members
2. member functions
3. nested types
    a) class
    b) struct
    c) union
    d) enum
    e) typedef declaration

Members are counted whatever visibility they have. Nested types can be anonymous. Members are not counted if they are friend classes or member functions.
Listing 3.6 illustrates an example:

```
class Example{
  int i;  // counts 1
public:
  class NestedExample1; // counts 1
  struct NestedExample2{  // counts 2 (itself as nested type and its member)
    int i;  // counts 1
  };
  enum{a,b,c};  // counts 1
  typedef int myInt;  // counts 1
  void doIt();  // counts 1
  friend class MyFried; // does not count
  friend void doAnother();  // does not count
};
void doNothing(); // does not count
```

Listing 3.6: Code snippet for Number of Members per Type

3.5. Efferent Coupling

Efferent Coupling is a metric that numbers on how many distinct foreign units the unit of interest relies on. Metriculator implements Efferent Coupling on type level. That means that the more foreign types a type depends on, the higher its efferent coupling value is. A dependency exists if a type is directly referenced.
Following Listing 3.7 illustrates an example:

```
class Example2;
class Example3;

class Example1 {
public:
  void doSomething(Example2 *e2) {};  // counts 1 for type Example1
  Example3 ex3();   // counts 1 to type Example 1
};

class Example2 {
public:
  void doSomething(Example1 *e1) {};  // counts 1 for type Example2
};

class Example3{
};
```

Listing 3.7: Code snippet for Efferent Coupling

4. Implementation

This chapter describes the main classes and concepts of the metriculator plug-in. First we give an overview of the package level architecture. Then we dive into the details of the model package by commenting static and dynamic aspects. At the end some performance related actions as well as known issues will be presented in detail.

4.1. Plug-in Architecture

As visible in Figure 4.1 metriculator is based on the Codan framework. Codan uses checkers to analyse source code. Each checker is specialised in one problem, for instance unreachable code. Through extension points Codan allows third party tools to register other checkers. Metriculator defines one checker per metric. As soon as a user invokes the Codan command on the UI, Codan automatically calls all registered checkers.
The metriculator component in Figure 4.1 can further be divided into the packages shown in Figure 4.2.

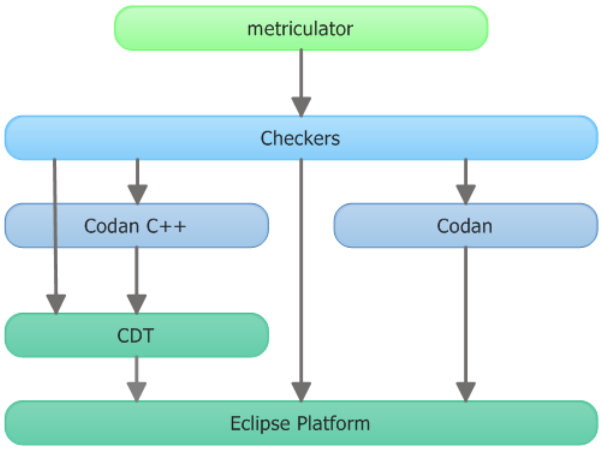


Figure 4.1.: Architectural representation of the components metriculator relies on.

The metriculator package is the core of this plug-in. Thanks to the *PluginActivator* we know when the Codan command is started and finished. This allows us to initialise the model each time before the checkers start working. After the work is done the *Plugin-Activator* notifies the views to update their data via observer. As the checkers run they build a tree model that represents the analysed files and their content.
The bidirectional dependency between the model and the metriculator component ex-

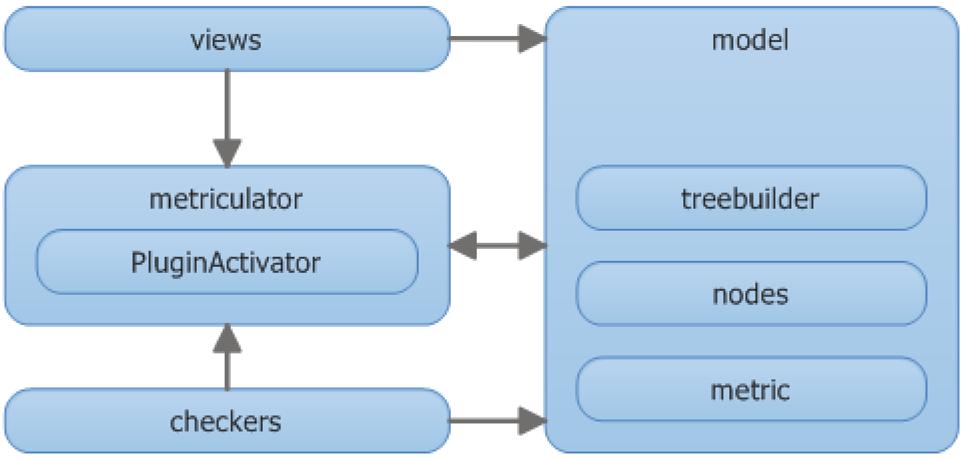


Figure 4.2.: Package diagram with packages inside the metriculator plug-in.

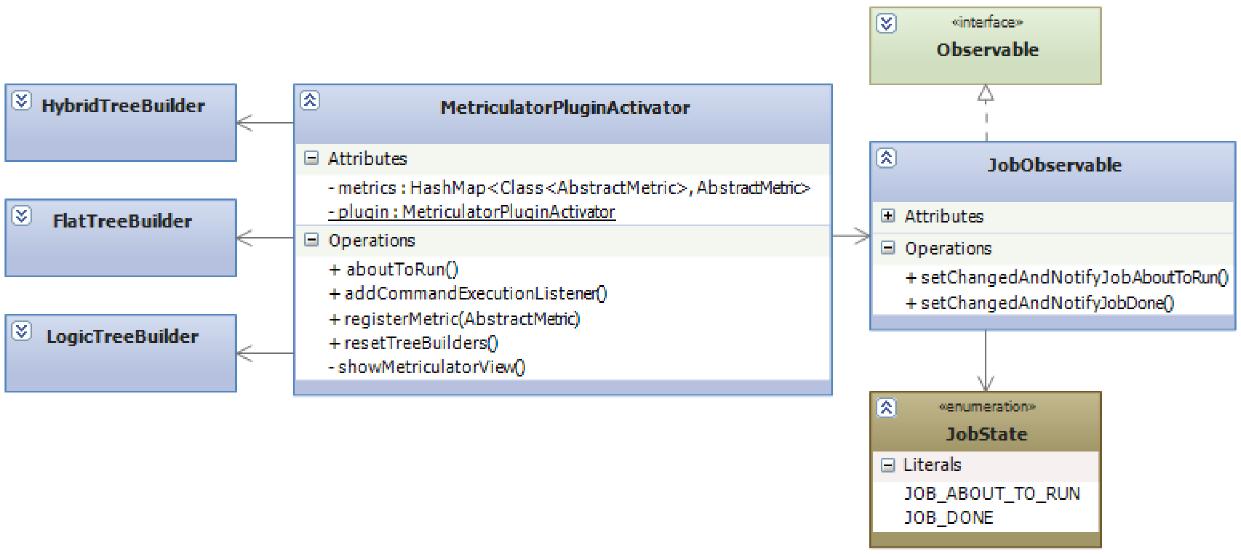
ists because some model classes use a singleton inside of the metriculator component. Singletons are used due to constraints of the Eclipse framework.

4.1.1. Plug-in Activator

The plug-in activator activates the plug-in when it is about to run. Every Eclipse plug-in inherits from the *AbstractUIPlugin* of Eclipse. The *MetriculatorPluginActivator* holds the singleton instance of this plug-in. The constructor of the *MetriculatorPluginActivator* registers itself at a job listener which notifies metriculator when Codan starts the code analysis and when Codan has done its job.
Before the metriculator plug-in starts its job, it has to reset all model data. The job listeners also reset the caches of the metric values, clear the views and models to dereference all old objects from a previous execution of metriculator. See 5.1 for further information.
All metric checkers has to register themselves at the *MetriculatorPluginActivater*. After Codan has finished its job, the *MetriculatorPluginActivator* invokes the aggregation of all values of the activated metrics and shows the metriculator view. *MetriculatorPluginActivator* has a *JobObservable* that implements the *Observable* and is responsible for notifying his observers if the plug-in is about to run or if its job is done. Observers of the *JobObservable* are the metriculator view and all metric checkers.

4.2. Tree Structure

This sections describes the composite tree structure that is build by the *AbstractMetricChecker* instances during the processing described in Chapter 4.4. This tree structure holds all the data shown in the metriculator view.



«interface»
Observable
HybridTreeBuilder
MetriculatorPluginActivator
Attributes
- metrics : HashMap<Class<AbstractMetric>, AbstractMetric>
- plugin : MetriculatorPluginActivator
Operations
+ aboutToRun()
+ addCommandExecutionListener()
+ registerMetric(AbstractMetric)
+ resetTreeBuilders()
- showMetriculatorView()
FlatTreeBuilder
LogicTreeBuilder
JobObservable
Attributes
Operations
+ setChangedAndNotifyJobAboutToRun()
+ setChangedAndNotifyJobDone()
«enumeration»
JobState
Literals
JOB_ABOUT_TO_RUN
JOB_DONE

Figure 4.3.: Class diagram with classes related to *MetriculatorPluginActivator*

4.2.1. TreeBuilder

As illustrated in Figure 4.4 metriculator uses three different tree models which correspond to the three different views. The hybrid tree model is used in the hybrid view, the logical tree model is used in the logical view and the flat tree model is used in the filter view.
The *TreeBuilder* class is an abstract factory that is used by the checkers to create tree models. *AbstractMetricChecker* uses a *HybridTreeBuilder* to add new nodes to the tree structure. During the processing only the hybrid tree model is build because the hybrid tree model holds all the information that other tree models require. Flat trees and logical trees are build on demand using the data of the hybrid tree model. For instance, when a user switches from the hybrid view to the logical or filter view. *FlatTreeBuilder* and *LogicalTreeBuilder* use a *PreOrderTreeVisitor* to visit all nodes in the hybrid tree model and transform it to the new respective tree model.
The class factory method *createTreeFromPath(IPath)* is used by the *HybridTreeBuilder* to create a tree structure that represents a file system path. To obtain a *LogicTreeBuilder* or *FlatTreeBuilder* instance clients use the *buildFrom(TreeBuilder)* method.

4.2.2. AbstractNode

As illustrated in Figure 4.5 the *AbstractNode* class is a simplified composite structure to hold all relevant data the view displays. Each node has one parent and any number of children. An *AbstractNode* represents a file system object or an *ASTNode*. The required information extracted from the *ASTNode* are hold in a *NodeInfo* instance. We extract the values from the *ASTNode* instead of holding a reference to it so that the *ASTNode*

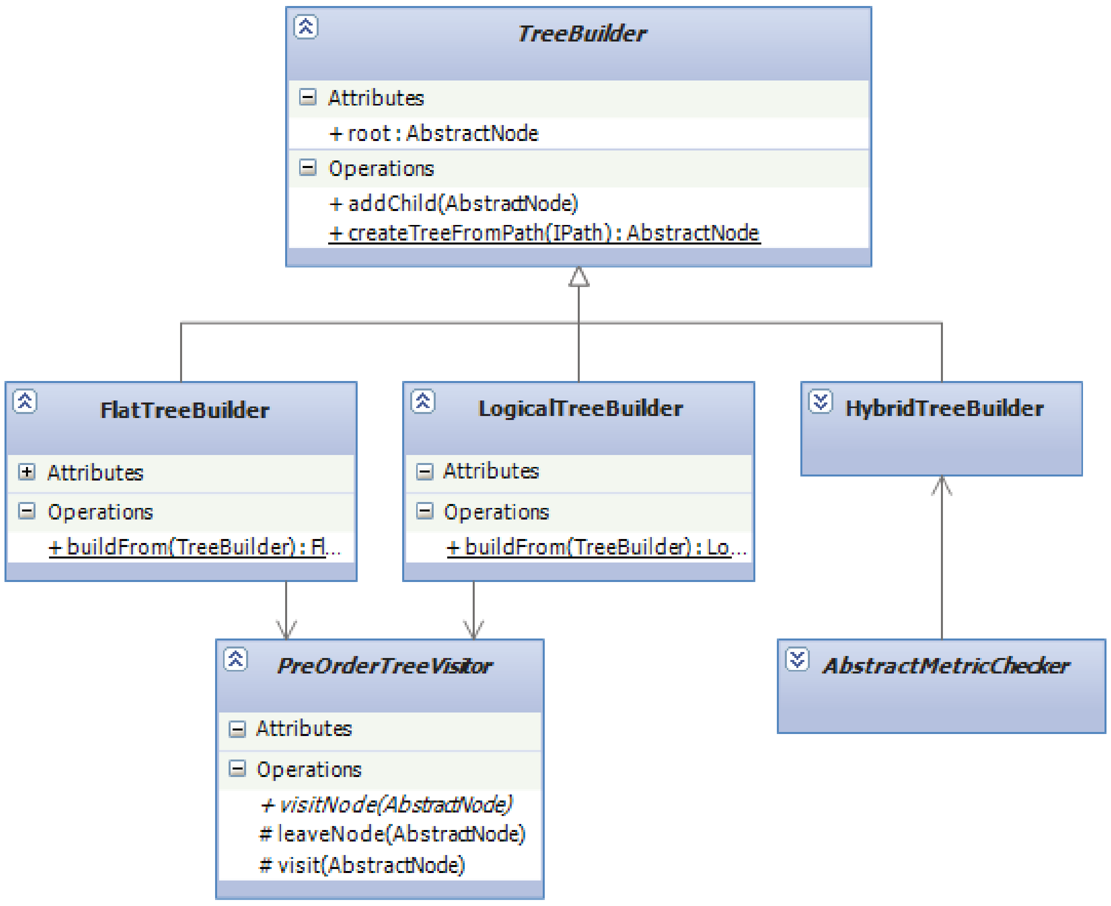


Figure 4.4.: Class diagram with classes related to *TreeBuilder*.

instance can be removed by the garbage collector. See chapter 5.1 for more details about related performance issues.

4.2.2.1. AbstractNode Identifiers

Each *AbstractNode* has three types of string values that name it: hybridId, scopeName, scopeUniqueName. All of them are illustrated in Figure 4.6.
The *hybridId* is the tree wide unique identifier of a node. It is needed to identify the nodes and building the tree structures. The *hybridId* is build by concatenating the *scopeUniqueNames* of all ancestor nodes and its own *scopeUniqueName*. To understand what a *scopeUniqueName* is we first explain what a *scopeName* is.
The *scopeName* is the name of the represented underlying object. For instance the file name or the signature of a function. Logical nodes do not always have a name. For instance anonymous namespaces may exist at the same level. Hence it would be a problem

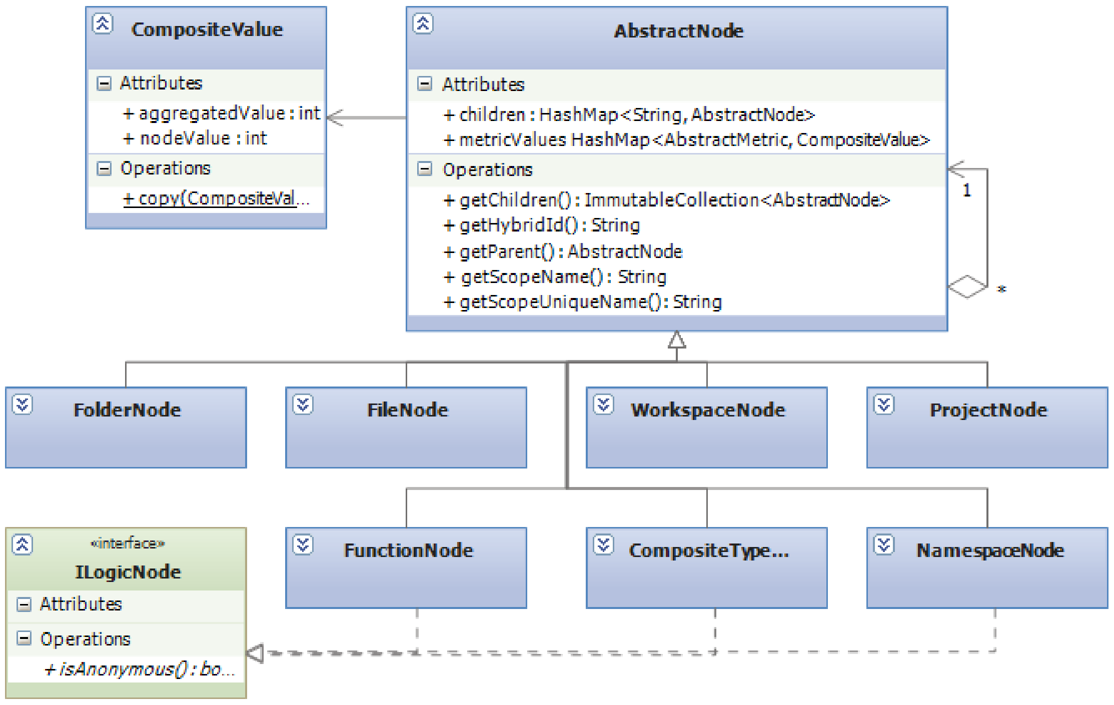


Figure 4.5.: Class diagram with classes related to *AbstractNode*.

to identify objects by using only their *scopeName* because two namespaces at the same level with no name would have the same *hybridId*.
Therefore we introduced the *scopeUniqueName*. The *scopeUniqueName* is build by appending the hash code of the *NodeInfo* instance to the existing *scopeName*.

Following example describes the identifier types used in *AbstractNode* and is illustrated in Figure 4.6:

- **scopeName** of function *f(int i)* is f(int i)
- **scopeUniqueName** of an anonymous namespace is just a number (hash code)
- **hybridId** of the function *f(int i)* therefore is
  Workspace:ProjectX:identifier_test.cpp:2470869:A1338645:f(int i)2730475

4.2.2.2. CompositeValue

The *CompositeValue* is used by *AbstractNode* objects to store the calculated values for each *AbstractMetric*. As shown in Figure 4.5 the *CompositeValue* is implemented as a pure value object.

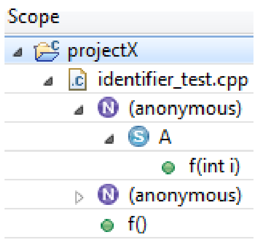
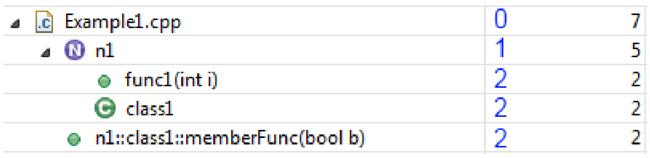


Figure 4.6.: Sample tree structure

The *nodeValue* field stores the current value of the node. The *aggregatedValue* field is calculated by the *nodeValue* of it self plus the sum of all *nodeValues* of its descendant nodes. The *aggregatedValue* is calculated once when the Codan job finishes. Calculation is done for all nodes in the *HybridTreeBuilder*. See chapter Performance Tuning 5.1 for more details. An example which describes the *CompositeValue* is illustrated in Figure 4.7:



Example1.cpp	0	7
n1	1	5
func1(int i)	2	2
class1	2	2
n1::class1::memberFunc(bool b)	2	2

Figure 4.7.: Example of *CompositeValue*, the values written in blue on the left are the node values and the other values on the right are the aggregated values.

4.2.3. TreePrinter

The *TreePrinter* class was mainly used for debugging purposes. We use it in our test cases to print the generated tree to the console after a test has been ran. Figure 4.8 shows a sample output. Scope names are trimmed to a maximum length and the metric values in each column are left aligned. The node values are in braces next to the aggregated values.

```
[metric columns: 'LSLOC']
 testworkspace                670 (0)
   testproject                670 (0)
     file1                    650 (10)
       ns1                    10 (10)
       ns31                   310 (310)
       ns32                   320 (320)
     file2                    20 (20)
```

Figure 4.8.: Sample output of the *TreePrinter* class.

4.3. Metric Checkers

Codan offers extension points for third party tools to add new checkers. Metriculator uses Codan's extension points to define one checker per software metric. Every checker in metriculator inherits from the *AbstractMetricChecker* that again inherits from the *AbstractIndexAstChecker* of Codan.
Metriculator uses the *AbstractMetric* class to encapsulate checkers and metric relevant data. Each checker implementation is associated with one *AbstractMetric* implementation. When the checker is created on application start up it associates itself with the corresponding metric instance. A list of all *AbstractMetric* instances is accessible through the plug-in singleton held by the *MetriculatorPluginActivator*. See Figure 4.9 for more details.

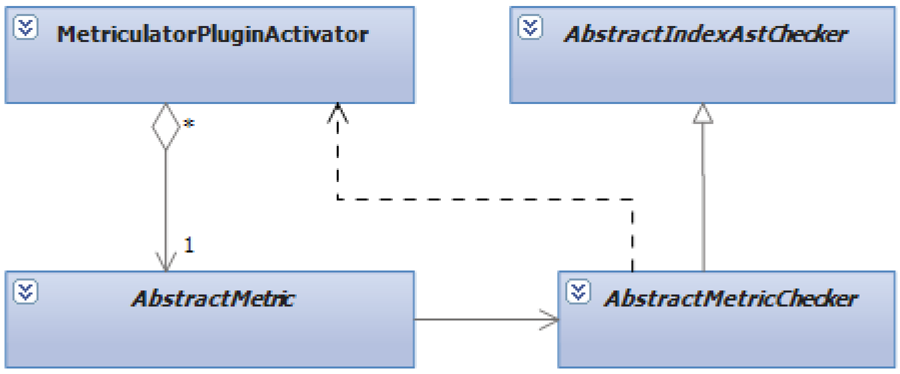


Figure 4.9.: Relations between the *MetriculatorPluginActivator*, *AbstractMetric* and *AbstractMetricChecker* classes.

Beside the checker reference *AbstractMetric* implementations are responsible to implement a value aggregation strategy. The *AbstractMetric* class provides a default aggregation strategy as you can see in Listing 4.1.

```
public int aggregate(AbstractNode node){
  CompositeValue metricValue  = node.getValueOrDefaultOf(getKey());
  metricValue.aggregatedValue = 0;

  for(AbstractNode child : node.getChildren()) {
    metricValue.aggregatedValue += aggregate(child);
  }

  metricValue.aggregatedValue += metricValue.nodeValue;

  return metricValue.aggregatedValue;
}
```

Listing 4.1: Code snippet of the default value aggregation strategy implemented in *AbstractMetric*

4.3.1. Tasks

Each *AbstractMetricChecker* implementation is responsible for the following tasks:

- Visiting of the AST using a *ScopedASTVisitor*. see 4.3.3
- Initialisation of its profile preferences
- Problem creation and reporting

The following tasks are taken care of by the *AbstractMetricChecker* class:

- Getting the hybrid tree model. see 4.2.1
- Creating the file and folder nodes for each processed file and adding it to he hybrid tree model. see also 4.4
- Resetting its problems, all tree models and *AbstractNode* references as soon as Codan has finished its job.

4.3.2. Scope Listeners

The *IScopeListener* interface allows metric checkers to register themselves at *ScopedAstVisitor* instances to be notified if the scope changes. Checkers for instance use this mechanism to report problems after function or type nodes have been analysed.
See Figure 4.10 to see the environment of the *AbstractMetricChecker* class.

4.3.3. AST Visitors

The *ScopedASTVisitor* is responsible for visiting the AST and has a reference to the current scope node which is a *AbstractNode* instance. The *ScopedASTVisitor* inherits from the *ASTVisitor* of Codan and implements the visit and leave methods. In the *ScopedASTVisitor* all relevant *ASTNodes* of a file, provided by the calling checker instance, are visited and added as nodes to the hybrid tree model. Thus all logical nodes of the hybrid tree model are created with *ScopedASTVisitor* classes.

ScopedASTVisitor implementations like *LSLOCScopedASTVisitor* are responsible for analysing the *ASTNodes* relevant for the related metric. *ScopedASTVisitor* create the node values for the specific metric in the current scope. There is always one *ScopedASTVisitor* associated with one *AbstractMetricChecker*. Visitors can also notify registered *IScopeListener* instances if a scope is changing. See chapter 4.3.2 for more details.

Figure 4.11 shows a class diagram of the *ScopedASTVisitor* and related classes.

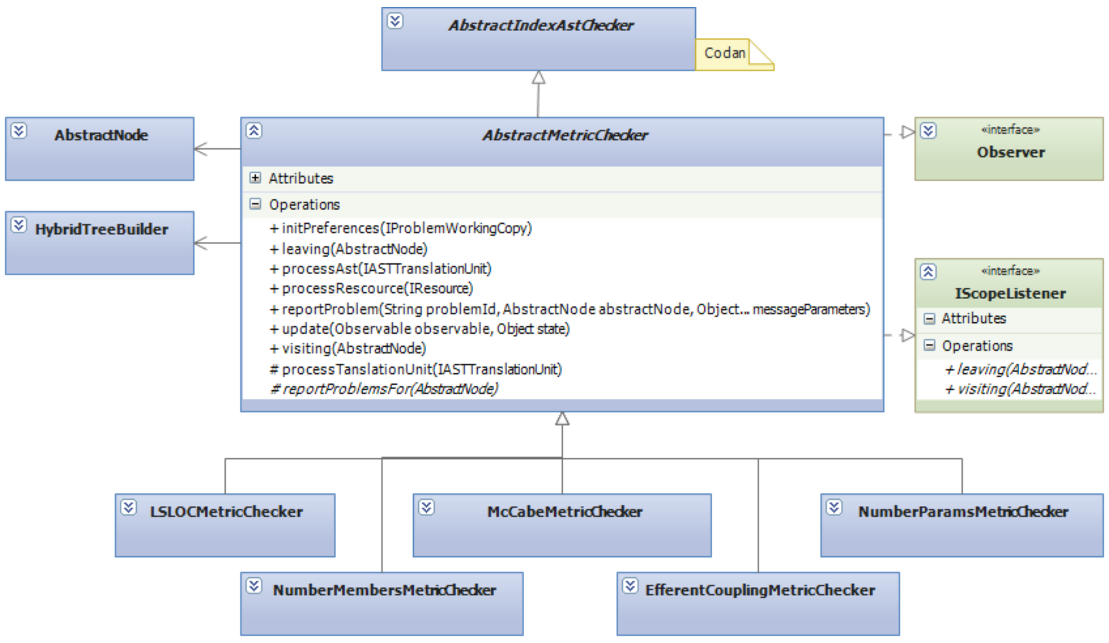

AbstractIndexAstChecker
Codan
AbstractNode
HybridTreeBuilder
AbstractMetricChecker
Attributes
Operations
+ initPreferences(IProblemWorkingCopy)
+ leaving(AbstractNode)
+ processAst(IASTTranslationUnit)
+ processRescource(IResource)
+ reportProblem(String problemId, AbstractNode abstractNode, Object... messageParameters)
+ update(Observable observable, Object state)
+ visiting(AbstractNode)
# processTanslationUnit(IASTTranslationUnit)
# reportProblemsFor(AbstractNode)
«interface» Observer
«interface» IScopeListener
Attributes
Operations
+ leaving(AbstractNod...
+ visiting(AbstractNod...
LSLOCMetricChecker
McCabeMetricChecker
NumberParamsMetricChecker
NumberMembersMetricChecker
EfferentCouplingMetricChecker

Figure 4.10.: Class diagram with classes related to *AbstractMetricChecker*

4.4. Processing

The Codan framework invokes the metriculator checkers. At the end of the processing of Codan, the view of metriculator appears. The whole processing is described in detail below and is illustrated in Figure 4.12.

1. A user runs Codan with the *Run C/C++ Code Analysis* command in Eclipse CDT.
2. Metriculator is notified about the start of Codan.
3. Metriculator resets its models.
4. Codan starts processing the C++ resources to be analysed.
5. For each resource each checker is called to process it.
6. Each translation unit is processed.
7. The metriculator checker visits the relevant AST nodes to calculate metric data.
8. When Codan has finished, metriculator shows its view in Eclipse.

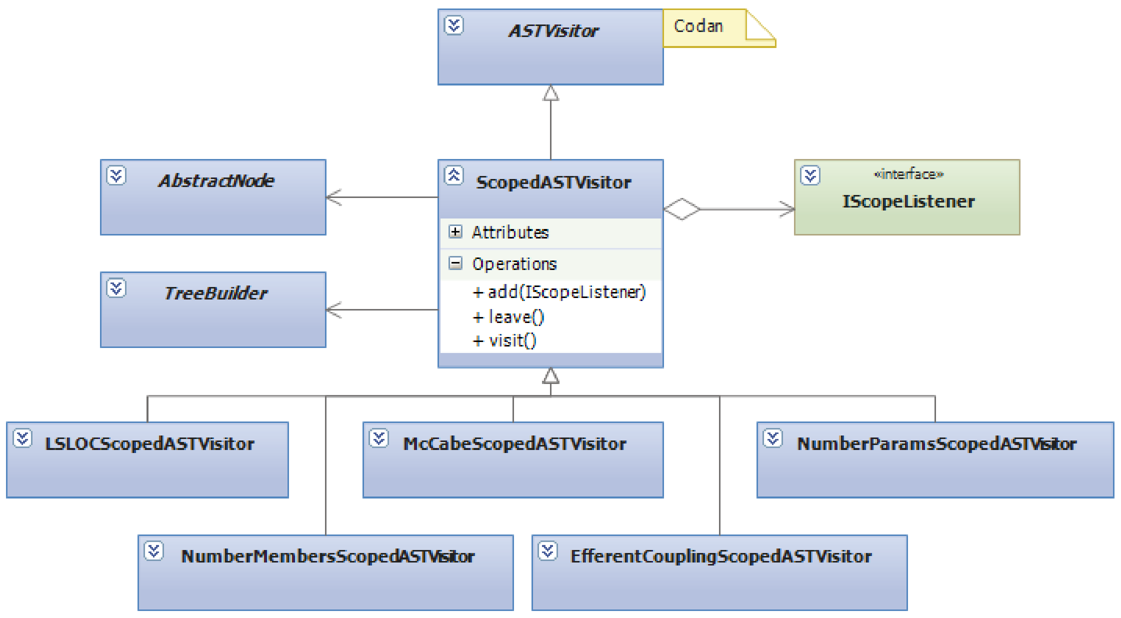


Figure 4.11.: Class diagram with classes related to *ScopedASTVisitor*

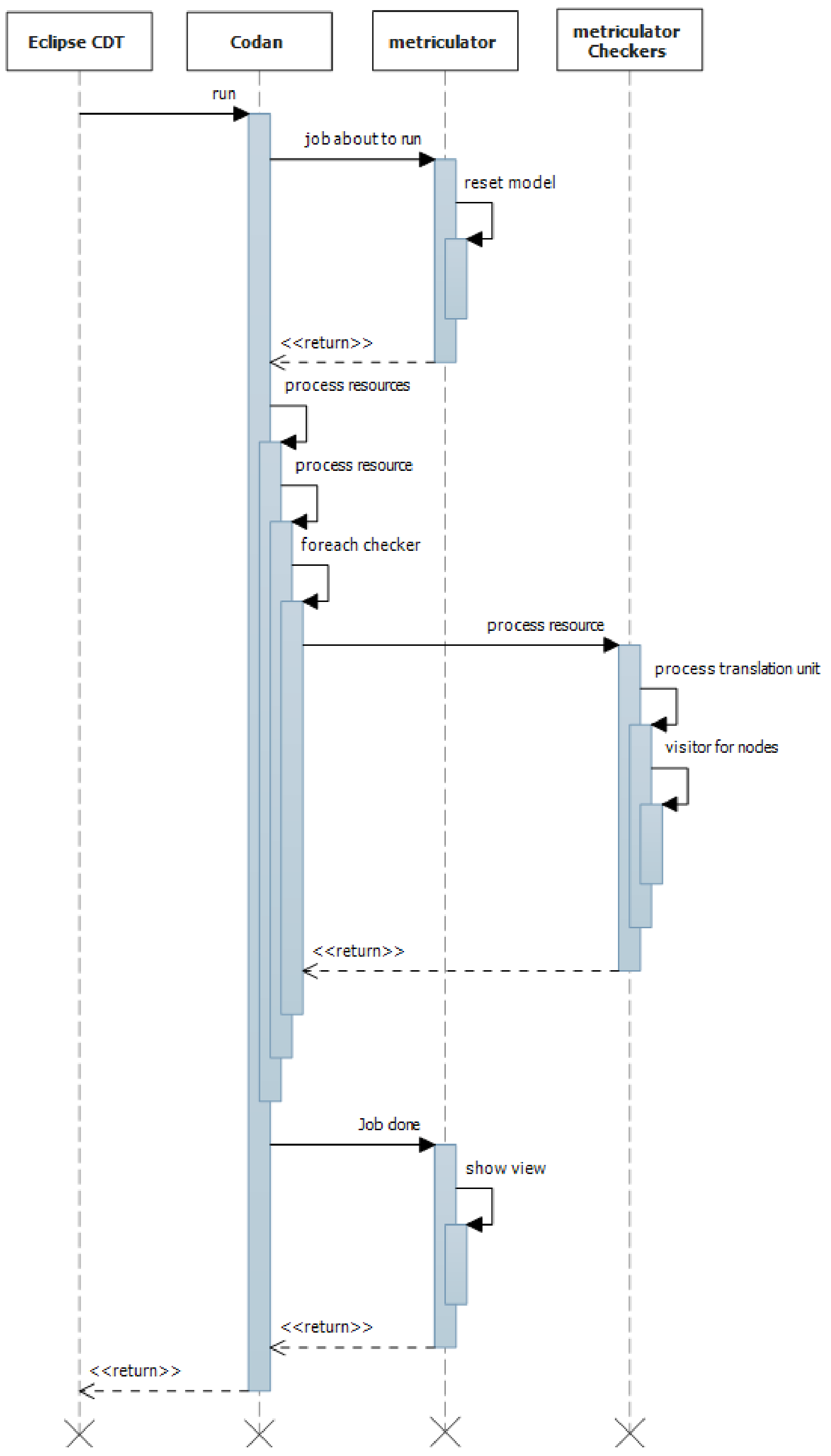


Figure 4.12.: High level sequence diagram of the processing of metriculator

5. Conclusion

Overall, we reached our basic objectives defined in chapter 2. The plug-in performs well in medium sized projects. It has a solid design and is easily extensible.
The following chapter describes analysis and problem finding processes related to performance issues. After that we describe the unresolved issues of the project. At the end we describe ideas to improve and extend metriculator.

5.1. Performance Tuning

In week nine we set up the last step in our deployment pipeline [HF10]. That is the continuous and automated publishing of the latest plug-in build. The plug-in was made available through a publicly available p2 update site [upd11]. The update site allowed us to install the plug-in on an independent eclipse instance which is mandatory to test the performance. According to the quantitative goals of this semester thesis 1 we ran our performance tests against the llvm project [llv11] especially the clang sub project [cla11].
The first manual test runs were very disappointing. The performance was very bad. Analysing two thousand LSLOC did work. Everything above was critical and most often eclipse ran out of heap space. Also the operation took very long to finish. We soon realized that for the remaining two weeks of development, performance tuning will have highest priority. The plug-in must perform well otherwise it would be useless, regardless of the number of features implemented. Thanks to jprofiler [jpr11] we quickly figured out the time consuming methods and memory bottle necks.

5.1.1. Timing Issues

We first optimized metriculator to take less time of execution. This was mainly achieved by reducing the number of method calls by caching values instead of recalculating them on every request.

An *AbstractNode* is able to sum up all node values of its descendant nodes. This process has been triggered very often by the tree viewer control since every cell displays such an aggregated value. To fix that problem we now only aggregate the values once at the end of the code analysis, before the data are bound to the view.
Another hot spot was the clone mechanism of an *AbstractNode*. Cloning was implemented as deep clone, which means that cloning one node in fact cloned the node itself and all descendant nodes. Before caching aggregation values this step was mandatory to show correct values in filter views. Thanks to cached values we changed the deep clone

implementation to a shallow clone implementation. That is much more faster and saves even more memory.
Similar to the value aggregation problem was the path algorithm problem. Each *AbstractNode* has a unique path that represents its position in the tree. The initial implementation requested the path of a parent node, every time a child was added. Since the path building algorithm was recursively it was very time and also memory consuming. Building the path only once and only force recalculation if mandatory solved that performance issue.

5.1.2. Memory Issues

The main reason why metriculator crashed at the beginning was because the huge amounts of memory allocation. The virtual machine run out of heap space and Eclipse crashed. We identified two problem areas.

During the processing metriculator created ten thousands of char[] instances that allocated mega bytes of memory. We figured out that the instances were hold by C++ parser class instances that were used to create the AST. Since *AbstractNode* referenced the *ASTNode* instances they could not be removed by the garbage collector.
As a solution we introduced a new class called *NodeInfo* that serves as pure value object holding only the relevant infos of an *ASTNode* that we really need to keep. As a result memory allocation significantly decreased.
But there is still capability left to improve the memory allocation in *NodeInfo*. We hold an *IBinding* reference that allows us to merge function declaration and function definition. The merging task has to run after all checkers ran, because function declarations and definitions may be in different files. Since the problematic memory allocation happens during the processing we can not release the *IBinding* references and that still causes a significant amount of memory being allocated.

The second problem was caused by the jface *TreeViewer* component. We used the *TreeViewer* component in the hybrid view, as well as in all others views, including the filter view. But in the filter view, we never displayed hierarchical data but very large lists of for instance *FunctionNodes*. The *TreeViewer* component is not optimized to display flat lists. Therefore we decided to use the *TableViewer* component in filter views. That decision tremendously improved the performance of the filter view mode. Now displaying and sorting thousands of *FunctionNodes* just takes a fraction of a second.

The introduction of the *TableViewer* component beside the existing *TreeViewer* component forced us to create duplicated, almost identical code. Although their API is very similar, *TreeViewer* and *TableViewer* do not share a common base that we could use to create an abstract interface to interact with both of them. Additionally subclassing is not allowed in SWT [swt11]. Therefore we just duplicated the code used for the *TreeViewer* and replaced the types used.

5.2. Known Issues

There are some known issues that should be taken care of in future releases. This chapter describes this issues and also contains a reference to the issues documented in Redmine.

5.2.1. Performance

As already described in detail in section 5.1 we optimized metriculator to perform better on large input data. Currently metriculator can analyse about 300'000 physical lines of code if problem reporting is disabled and only the LSLOC metric is active.

Issue #117 at http://sinv-56013.edu.hsr.ch/redmine/issues/117

5.2.2. Zest Cloudio Integration

The tag cloud integration is not stable yet. The code of this feature was taken from the project at [sou11] after the author contacted us and suggested to integrate his work. We just customized the code to work within metriculator. Generating a tag cloud based on large lists (for instance function nodes) sometimes results in exceptions saying that there is not enough drawing space available.

Issue #176 at http://sinv-56013.edu.hsr.ch/redmine/issues/176

5.2.3. Installation via Composite Update Site

When installing metriculator multiple steps are required in advance to install the prerequisites. Before metriculator can be installed via update site, users have to install CDT and the Zest framework separately. The current composite update site does not work.

Issue #177 at http://sinv-56013.edu.hsr.ch/redmine/issues/177

5.2.4. Merging of Function Declarations in Anonymous Namespaces

This bugs only relates to the logical view. If a function is declared in an anonymous namespace and the definition is outside of the anonymous namespace the definition does not replace the declaration. Instead the declaration and the definition are deleted.

Issue #166 at http://sinv-56013.edu.hsr.ch/redmine/issues/166

5.3. Future Work

There are many interesting not yet implemented metrics that would increase the value of metriculator, see chapter 1.2.1 for an incomplete list.
Performance can further be improved to allow analysing source code with over about 300'000 physical lines of source code. Fixing that issue would make metriculator even

more attractive to analyse large projects. Other unresolved issues are listed in chapter 5.2.
A reasonable application of metriculator in the future is to serve as refactoring assistant. Based on a reported problem, metriculator could suggest a refactoring or quick fix that solves that problem.

A. Environment Set up

This appendix describes the hardware and software components that support us in reaching our project goals. We give detailed installation and configuration instructions and highlight problem areas to be aware of when setting up a similar environment.

A.1. Hardware

We use a virtual server to host different kinds of software that support us in our daily project tasks. The virtual server is hosted by the HSR. We have full root access and can connect to the server by VPN if we are outside of the *HSR-LAN*. The server runs with Ubuntu 10.04 TLS on 1GB RAM. The host name is `sinv-56013.edu.hsr.ch`.

A.2. Project Management Software

To support our project management tasks we decided to use Redmine. The latest release at the start of our project was version 1.2.1. We use Redmine for the following tasks:

- issue tracking
- time tracking
- meeting records
- Git repository browsing (login required)
- Gather and share thoughts

Our Redmine instance is publicly and read only available at `http://tiny.cc/metriculator`.

A.2.1. Set up & Configuration

We followed the set up instructions on [red11] to install and configure Redmine using Passenger.
Basically we used the following commands:

```
sudo aa-complain /usr/sbin/mysqld
sudo apt-get update
sudo apt-get install redmine redmine-mysql
sudo apt-get install libapache2-mod-passenger
sudo ln -s /usr/share/redmine/public /var/www/redmine
sudo a2enmod passenger
sudo chmod a+x /usr/share/redmine/public
sudo service apache2 restart
```

Listing A.1: Redmine set up commands

After these steps, Redmine is up and running at `http://host/redmine`.
To enable Redmine repository browsing the user under which Apache runs has to have read access to the Git root directory. By default Apache runs as *www-data*. Additionally, you might have to refresh Redmines repository cache.

```
# preferably gather all users that require git access in one group
sudo adduser www-data group-with-read-access
cd /path/to/redmine && /usr/bin/ruby1.8 script/runner \
"Repository.fetch_changesets" -e production
```

Listing A.2: Redmine repository browsing set up

Caution: Redmine version 1.2.x requires rails version 2.3.11 (with Ruby gems 1.3.7). We also had to reset our MySQL root password. We followed the instructions on [mys11] to accomplish this.

A.3. Version Control System, Git

To support our file version management we decided to use Git. The latest release at the start of our project was version 1.7.6. We used the following commands to set up Git:

```
# login as root

#add a group for the git-users
group="git-users"
groupadd "$group"

# add a user
user="name_of_user"
adduser "$user"

# add the user to the git group
usermod -aG "$group" "$user"

# shared directory for the git-repository
mkdir -pv /var/gitrepo
chmod 770 /var/gitrepo
chgrp "$group" /var/gitrepo
chmod g+s /var/gitrepo

# create a git repository
# --shared: all users of the group must have access
# --bare: no working directory, it is just the repository
cd /var/gitrepo
```

```
mkdir repository.git
cd repository.git

git init --bare --shared=group

# acces via ssh with public-key authentication
# every user has to generate a key
# linux: generates a private and a public key in ida_rsa.pub
ssh-keygen -t rsa -b 2048 -f ~/.ssh/id_rsa

# the public key of the users has to be installed on the server
umask 077
mkdir -v "/home/${user}/.ssh"
cat "id_rsa_${user}.pub" >> "/home/${user}/.ssh/authorized_keys"
chown "$user:$user" "/home/${user}/.ssh"
# "ida_rsa_${user}.pub" ist he public key of the user

# cloning the git repository
git clone ssh://user@host/var/gitrepo/repository.git
```

Listing A.3: Git repository set up

Caution: For using Git on Windows and how to set up the required SSH access see [win11].

A.4. Development Environment

The plug-in was developed in Eclipse Indigo using the plug-in development environment (PDE) plug-in. To test metriculator we used sample C++11 source code. To force the compiler to build according to the C++11 standard add the `-std=gnu++0x` flag to the following field: *Project Properties > C/C++ Build > Settings > GCC C++ Compiler > Miscellaneous > Other flags.*

A.5. Continuous Integration Server

We installed Jenkins 1.434 [jen11] as continuous integration server. Jenkins provides a solid platform for various plug-ins that enhance its basic continuous integration features. Our Jenkins instance requires the plug-ins listed in table A.1.
With Jenkins in place we can always monitor the health of our latest commits. For example if some unit tests fail, Jenkins will report that on the project homepage.

A.5.1. Set up & Configuration

To set up Jenkins we followed the set up instructions on [jen11]. Basically we used the following commands:

Name	Version	Description
Maven 2 Project	1.430	Allows to trigger Maven goals on build events.
Static Analysis Utilities	1.30	Provides utilities for the static code analysis plug-ins.
Static Analysis Collector	1.17	This plug-in is an add-on for the plug-ins Checkstyle, Dry, FindBugs, PMD, Tasks, and Warnings: the plug-in collects the different analysis results and shows the results in a combined trend graph.
FindBugs plug-in	4.29	This plug-in collects the FindBugs analysis results of the project modules and visualizes the found warnings.
Hudson Xvnc	1.10	Allows projects to run Xvnc during a build. Xvnc is required to run CDT plug-in tests since they start Eclipse.
Jenkins GIT	1.1.12	Integrates Git with Jenkins.
Jenkins Emma	1.25	Integrates EMMA code coverage reports to Jenkins.
ChuckNorris	0.4	Displays a picture of Chuck Norris followed by enlightening statements.

Table A.1.: Installed Jenkins plug-ins

```
sudo aptitude install openjdk-6-jre
wget -q -O - http://pkg.jenkins-ci.org/debian/jenkins-ci.org.key | \
sudo apt-key add -
sudo sh -c 'echo deb http://pkg.jenkins-ci.org/debian binary/ > \
/etc/apt/sources.list.d/jenkins.list'
sudo aptitude update
sudo aptitude install jenkins
```

Listing A.4: Jenkins set up commands

A.6. Build and Deployment Automation

The CDT project supports ant and Maven as build automation platform. We decided to give Maven (in contribution with Tycho) a try because it seemed a lot easier to maintain and has already been used in recent projects at HSR.

A.6.1. Set up & Configuration for Windows

Since we have never used Maven before, we started using Maven on the local development environment. It was not mandatory to do something locally, but it simplified to get in touch with Maven for the first time. We first set up the initial configuration files for our projects. We followed the following steps to set up Maven on our Windows development machines:

- Download Maven from [mav11a] and extract archive to <target>
- Add <target>\bin to the PATH variable
- Open your command line, change to your projects root directory.
- Run
  `run mvn org.codehaus.tycho:maven-tycho-plugin:generate-poms`
  `-DgroupId=ch.hsr.ifs.cdt.metriculator`
  `-Dtycho.targetPlatform=C:\Programme\eclipse`
  Where targetPlatform should point to your Eclipse installation folder.
  This command will generate or update pom.xml files in all directories in your project root. Chapter A.6.3 explains how to customize this pom.xml files.

A.6.2. Set up & Configuration for Ubuntu

Follow the steps on [mav11b] to install Maven 3 on Ubuntu. Basically these are the steps:

```
sudo mkdir /usr/local/apache-maven
cd /usr/local/apache-maven/
sudo wget http://ftp.heanet.ie/mirrors/www.apache.org/dist/ \
maven/binaries/apache-maven-3.0.2-bin.tar.gz
tar -xzvf apache-maven-3.0.2-bin.tar.gz
```

Listing A.5: Maven set up commands

After that make sure to configure the environment variables as described at [mav11b].

Our plug-in unit tests require Maven to execute headless builds. That for we have to install Xvnc with a X11 window manager. Use the following commands to install Xvnc as virtual screen buffer and metacity as window manager.

```
sudo aptitude install vnc4server metacity
sudo su jenkins
vncserver
# enter password manually
```

Listing A.6: Set up xvnc

After the install command we have to manually set the vncserver password for the Jenkins user. Otherwise the Jenkins build will fail because no password has been set.

A.6.3. Maven XML Configuration

Maven uses pom.xml for build instructions. We have one pom.xml in the root directory of all Eclipse projects (root pom) and one in each subdirectory (project pom). To automatically generate an initial version of the pom files follow the instructions in chapter A.6.1. All pom files are checked-in to the VCS as well. This chapter only highlights the changes applied to the generated pom files.

Version The version tag in all pom files should specify the same version as your plug-in manifest does. We use `<version>0.0.1.qualifier</version>` in the project pom files and `Bundle-Version: 0.0.1.qualifier` in our plug-in manifests.
All project pom files refer to the root pom file. The reference contains a version tag as well. The version tag of the root pom and the version referenced in the project pom files have to match. We use `<version>0.0.1-SNAPSHOT</version>`.

Repositories Although we imported some CDT projects into our Eclipse workspace (see chapter A.4) there is no need for Maven to build them all. The only modules that Maven has to build are the metriculator projects. Maven knows how to resolve the dependencies of our projects and may download them from remote repositories. We only need to specify the repositories to search in our root pom.

```
<repositories>
       <repository>
         <id>cdt-indigo</id>
     <!-- required to resolve cdt.core.tests dependencies of test project
     (in eclipse we use the project 'testing-project'
     that provides required jar files) -->
         <url>http://download.eclipse.org/tools/cdt/updates/indigo</url>
         <layout>p2</layout>
       </repository>
     <repository>
         <id>indigo</id>
         <url>http://download.eclipse.org/releases/indigo</url>
         <layout>p2</layout>
      </repository>
    <repository>
         <id>updates</id>
         <url>http://download.eclipse.org/eclipse/updates/3.7</url>
         <layout>p2</layout>
```

```
        </repository>
        <repository>
         <!-- indigo swtbot release not available yet, 14.nov.2011 -->
             <id>swtbot</id>
             <url>http://download.eclipse.org/technology/swtbot/helios/ \
                  dev-build/update-site</url>
             <layout>p2</layout>
           </repository>
         <repository>
             <id>zest</id>
             <layout>p2</layout>
             <url>https://hudson.eclipse.org/hudson/job/gef-zest-integration \
             /ws/org.eclipse.zest.repository/target/repository/</url>
          </repository>
      </repositories>
```

Listing A.7: Maven repositories

Caution: Note that at the time of writing the latest swtbot release available is the one for helios.

Target Platform Since we want Maven to be able to build on Windows machines as well as on our Linux based CI server we had to define different target platforms in our root pom. We use Tycho in version 0.10.0.

```
    <groupId>org.sonatype.tycho</groupId>
    <artifactId>target-platform-configuration</artifactId>
    <version>${tycho-version}</version>
    <configuration>
       <resolver>p2</resolver>
       <environments>
         <environment>
            <os>win32</os>
        <ws>win32</ws>
        <arch>x86</arch>
         </environment>
         <environment>
        <os>linux</os>
        <ws>gtk</ws>
          <arch>x86_64</arch>
         </environment>
      </environments>
    </configuration>
```

Listing A.8: Maven target platform configuration

A.6.4. Jenkins Maven Integration

There is a Maven plug-in available for Jenkins. This plug-in wraps the `mvn` command of Maven. But it is also possible to use a custom build script that invokes the `mvn` command. Since our build requires other actions being executed prior to the Maven build we use a custom build script. Another advantage of such a script is, that it is easier to maintain. All build steps are saved in one place and easy to change.

```
#!/bin/sh

export PATH=/usr/local/apache-maven/apache-maven-3.0.3/bin:$PATH
#make $DISPLAY (from xvnc) globally available so that
#other processes can access it.
export DISPLAY=$DISPLAY
THIS=$(readlink -f $0)
BUNDLE_ROOT="`dirname $THIS`"

cd $BUNDLE_ROOT
mvn -e clean install
```

Listing A.9: Jenkins build script

B. Metriculator Metrics

This chapter describes the results of a metric analysis applied to the metriculator plug-in source code.

B.1. Finding Problems using FindBugs

FindBugs did not find any problems.

B.2. Static Source Code Analysis

We used the Eclipse metrics plug-in [met11a] to analyse the metriculator source code.

B.2.1. Warnings

The analysis found two warnings, both are related to the McCabe metric.
The method *LSLOCScopedASTVisitor.visit(IASTStatement)* has a McCabe value of 16 which is 6 above the threshold of 10. Since this method has already been refactored heavily and reviewed by our supervisor we decided not to change the current implementation.
The method *NodeInfo.equals(Object)* has a McCabe value of 14 which is 4 above the threshold of 10. Since we have many fields to compare the McCabe value is exceptionally high. Based on the best practice patterns [jia11] we used in our implementation we do not think it would make much sense to further refactor the method.

B.2.2. metriculator Metrics Applied to Itself

This section outlines the metric results of the metrics that both the metrics plug-in and metriculator implement. The metrics plug-in does not always use the exact same metric names as metriculator does, but they can easily be mapped to the names we used.

Method Lines of Code (LSLOC) Beside the code in the tag cloud component, which we did not change (details at 5.2.2), the method *MetriculatorPluginActivator.showMetriculatorView()* has the most lines of code, namely 40. This is because we create an anonymous class that implements an interface with plenty of methods. Since there is no adapter for that interface we have to implement all the methods, although we just need one. The test package has 1410 lines of code. All other packages (views, resources, checkers, metriculator, model.*) together count 3559 lines of code.

Efferent Coupling The highest efferent coupling on package level is 15, for the package ch.hsr.ifs.metriculator.checkers. This package has many references to the model packages.

Number of Parameters The highest value is 4 which is good enough.

Number of Attributes (Number of Members per Type) The class *MetriculatorView* has the highest value of 22.

B.3. Dependecy Analysis

We used Structure101 to analyse the architecture of metriculator. As visible in Figure B.1 Structure101 detected eleven cyclomatic dependencies between the model and the metriculator package. Nine out of this eleven dependencies are because the *AbtractMetricChecker* class requires to use the plug-in singleton. The remaining two dependencies also come from the *AbstractMetricChecker* class. Metric checkers register itself as job listener to be notified when the Codan command is about to run or has finished. This events are mandatory to trigger for example post execution actions like value aggregation.

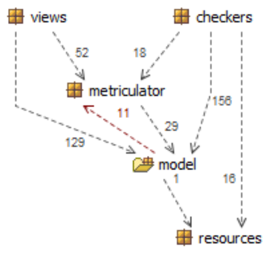


Figure B.1.: Dependency graph of metriculator generated by Structure101.

B.4. Test Coverage

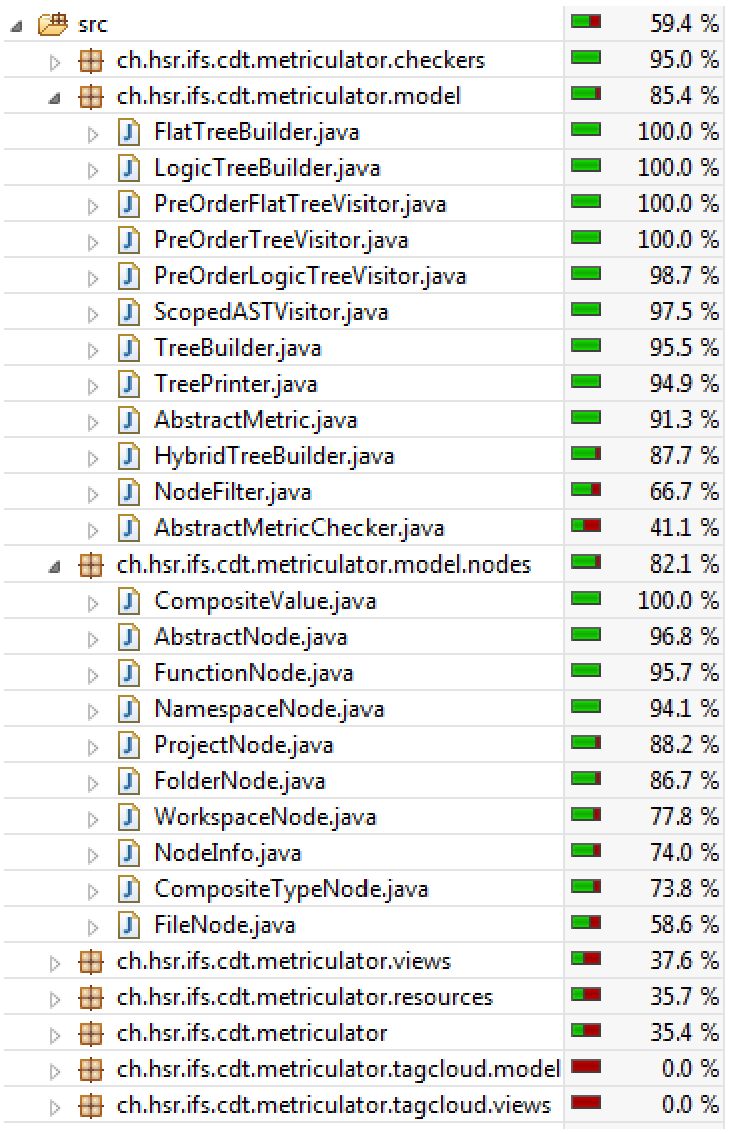
We used the EclEmma plug-in for Eclipse [emm11] to analyse the code coverage after running all unit test. The coverage results are shown in Figure B.2. The left column contains the packages and files analysed, the right column numbers the test coverage in percent. The elements of our interest are expanded and will be explained in this section.

B.4.1. UI Code

The packages mainly dedicated to the view have a low test code coverage. This is because we do not have ui test code. Following packages heavily contain code that is related to the view: views, resources, tagcloud.*.
The *AbstractNode* implementations in the package model.nodes all have a coverage of over 50%. The reason for this low coverage in some classes is because they all implement the *AbtractNode.getIconPath()* method which is only called from the view. Also the *AbstractMetricChecker* has 7 out of 15 methods that are only called if the view is run.

B.4.2. Model Code

The checkers and model packages have a test coverage of over 80%. Most units tests validate the correctness of the checker implementations which is reflected in a 95% coverage of the checker package.



src	59.4 %
ch.hsr.ifs.cdt.metriculator.checkers	95.0 %
ch.hsr.ifs.cdt.metriculator.model	85.4 %
FlatTreeBuilder.java	100.0 %
LogicTreeBuilder.java	100.0 %
PreOrderFlatTreeVisitor.java	100.0 %
PreOrderTreeVisitor.java	100.0 %
PreOrderLogicTreeVisitor.java	98.7 %
ScopedASTVisitor.java	97.5 %
TreeBuilder.java	95.5 %
TreePrinter.java	94.9 %
AbstractMetric.java	91.3 %
HybridTreeBuilder.java	87.7 %
NodeFilter.java	66.7 %
AbstractMetricChecker.java	41.1 %
ch.hsr.ifs.cdt.metriculator.model.nodes	82.1 %
CompositeValue.java	100.0 %
AbstractNode.java	96.8 %
FunctionNode.java	95.7 %
NamespaceNode.java	94.1 %
ProjectNode.java	88.2 %
FolderNode.java	86.7 %
WorkspaceNode.java	77.8 %
NodeInfo.java	74.0 %
CompositeTypeNode.java	73.8 %
FileNode.java	58.6 %
ch.hsr.ifs.cdt.metriculator.views	37.6 %
ch.hsr.ifs.cdt.metriculator.resources	35.7 %
ch.hsr.ifs.cdt.metriculator	35.4 %
ch.hsr.ifs.cdt.metriculator.tagcloud.model	0.0 %
ch.hsr.ifs.cdt.metriculator.tagcloud.views	0.0 %

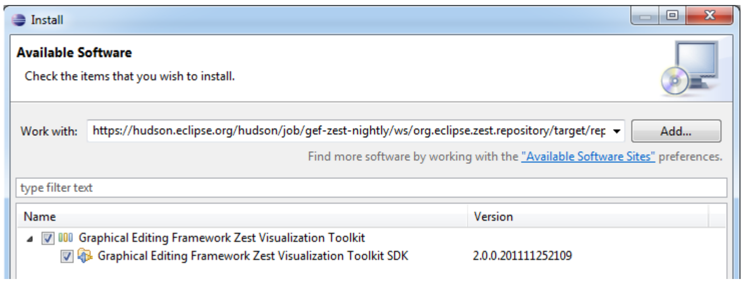
Figure B.2.: Screenshot of the emma code coverage analysis after running all unit tests.

C. User Manual

This user manual explains developers familiar with Eclipse and C++ how to use metriculator. It assumes that Eclipse CDT is already installed.

C.1. Installation

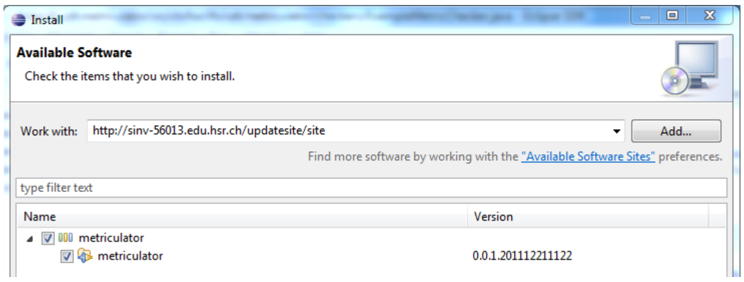
1. Install the Zest framework via update site from:
   https://hudson.eclipse.org/hudson/job/gef-zest-integration/ws/org.eclipse.zest.repository/target/repository/ (see Figure C.1)


Install
Available Software
Check the items that you wish to install.
Work with: https://hudson.eclipse.org/hudson/job/gef-zest-nightly/ws/org.eclipse.zest.repository/target/rep
Find more software by working with the "Available Software Sites" preferences.
type filter text

Name	Version
☑ Graphical Editing Framework Zest Visualization Toolkit	
☑ Graphical Editing Framework Zest Visualization Toolkit SDK	2.0.0.201111252109

Figure C.1.: Use the wizard at *Help > Install New Software ...*

2. Install metriculator via update site from:
   http://sinv-56013.edu.hsr.ch/updatesite/site (see Figure C.2)


Install
Available Software
Check the items that you wish to install.
Work with: http://sinv-56013.edu.hsr.ch/updatesite/site
Find more software by working with the "Available Software Sites" preferences.
type filter text

Name	Version
☑ metriculator	
☑ metriculator	0.0.1.201112211122

Figure C.2.: Use the wizard at *Help > Install New Software ...*

C.2. Start code analysis with metriculator

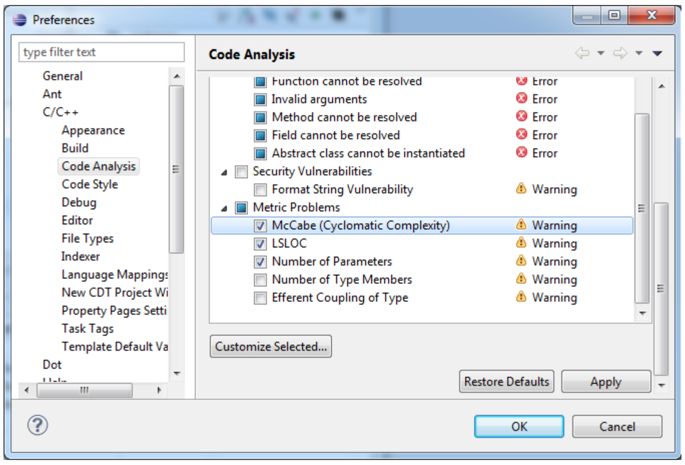
1. Choose your C++ source code to be analysed. Select one or more files and folders.

2. In the Project Explorer right click on the selection. This can be one or more projects, folders and files. Then choose to run the command *Run C/C++ Code Analysis*. This will start the static code analysis and metriculator will analyse the source code with the activated metrics. After the analysis the metriculator view opens.

C.3. Configuration of metriculator

1. Open the Eclipse preferences via *Window > Preferences*

2. Open the Code Analysis preferences for C/C++ (see Figure C.3)



■ Function cannot be resolved	Error
■ Invalid arguments	Error
■ Method cannot be resolved	Error
■ Field cannot be resolved	Error
■ Abstract class cannot be instantiated	Error
☐ Security Vulnerabilities	
☐ Format String Vulnerability	Warning
■ Metric Problems	
✓ McCabe (Cyclomatic Complexity)	Warning
✓ LSLOC	Warning
✓ Number of Parameters	Warning
☐ Number of Type Members	Warning
☐ Efferent Coupling of Type	Warning

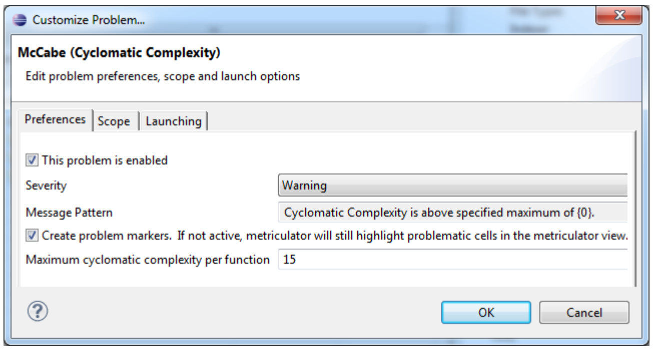
Figure C.3.: metriculator checker preferences

3. Under *Metric Problems* you can enable or disable metrics you want to run (see Figure C.3)

4. Double click on a metric to open the metric preferences dialog.

5. Specify the thresholds for the metrics (see Figure C.4)

6. Activate / Deactivate problem reporting with problem markers (see Figure C.4). If you deactivate the problem reporting, the metriculator view will still highlight problematic cells, but the problem will not be reported to the Eclipse *Problems View*. That implies that no markers are created in source code editors.


Customize Problem...

McCabe (Cyclomatic Complexity)

Edit problem preferences, scope and launch options

☑ This problem is enabled

Severity: Warning

Message Pattern: Cyclomatic Complexity is above specified maximum of {0}.

☑ Create problem markers. If not active, metriculator will still highlight problematic cells in the metriculator view.

Maximum cyclomatic complexity per function: 15

Figure C.4.: Customise Problem

C.4. Working with the Views

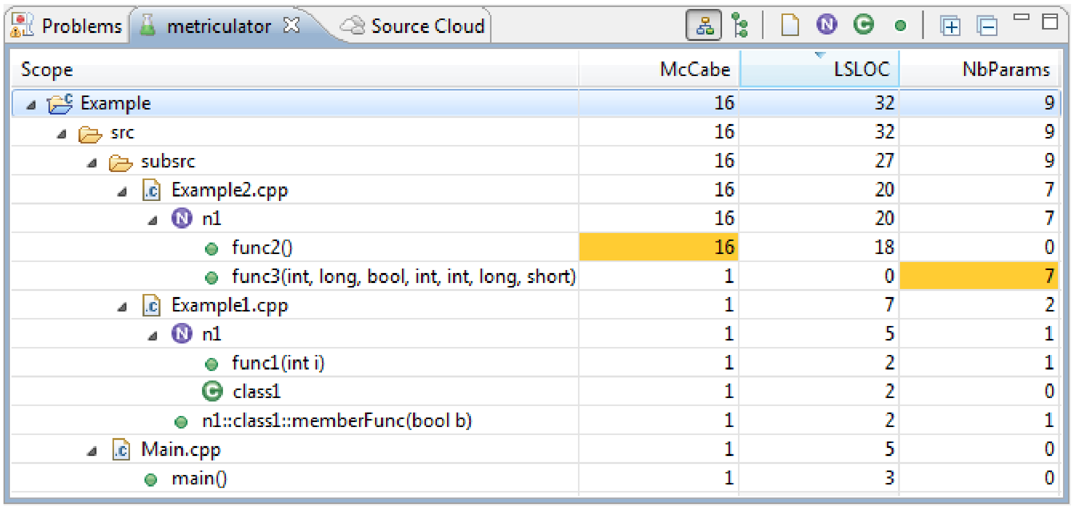
Use the different views to watch at the results. That helps you detect problems and obtain an overview of the quality and size of the analysed source code. Different metriculator views show the metric values from different perspectives. In all views all columns can be sorted. Cells with metric values beyond the thresholds are highlighted, this helps figuring out potential problems in your source code. All scope labels, for example a function name, can be double clicked to open an eclipse editor with the file where the function is in and to select the function code. Views with a tree view (hybrid view and logical view) can be easily expanded and collapsed with the commands in the upper right corner of the metriculator view.

C.4.1. View Types

- hybrid view (Figure C.5)
- logical view (Figure C.6)
- filter view (Figure C.7)
- tag cloud (Figure C.8)

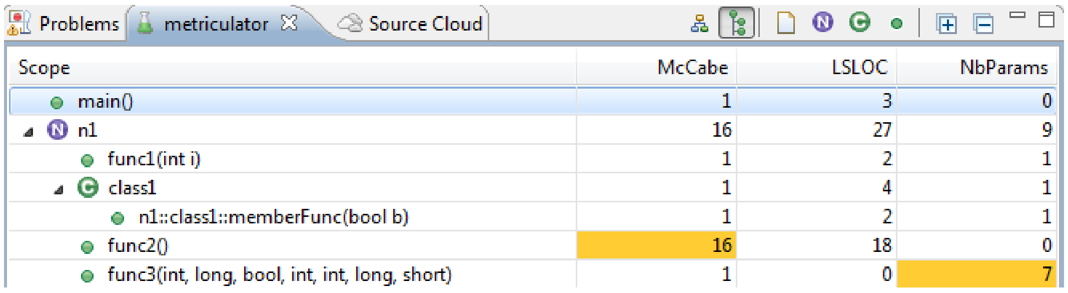
C.5. Problem Reporting and Markers

Metriculator allows to report problems to the Problems View built-in Eclipse. This will also set markers in source code editors next to the problematic source code sections. If the option *Create problem markers* in the metric problem preferences is activated (see Figure C.4), metriculator reports the problems (Example C.9) and sets the markers (Example C.10). Otherwise the problems are just visible inside the metriculator view. Cells with problematic values are highlighted anyway.
Hint: Deactivating problem reporting can improve the speed of the analysis. [ecl11a].



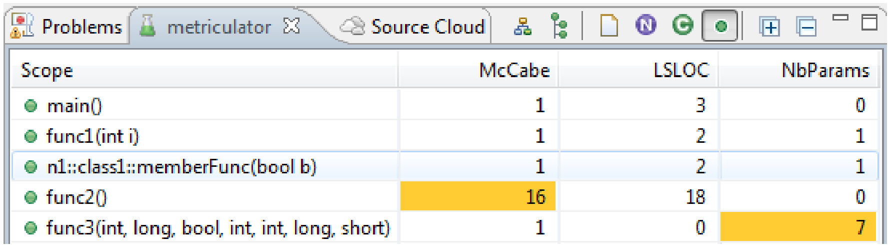
Scope	McCabe	LSLOC	NbParams
Example	16	32	9
src	16	32	9
subsrc	16	27	9
Example2.cpp	16	20	7
n1	16	20	7
func2()	16	18	0
func3(int, long, bool, int, int, long, short)	1	0	7
Example1.cpp	1	7	2
n1	1	5	1
func1(int i)	1	2	1
class1	1	2	0
n1::class1::memberFunc(bool b)	1	2	1
Main.cpp	1	5	0
main()	1	3	0

Figure C.5.: Hybrid view - This is the default view. It shows the projects and its content as a mix of physical and logical nodes in a tree structure.



Scope	McCabe	LSLOC	NbParams
main()	1	3	0
n1	16	27	9
func1(int i)	1	2	1
class1	1	4	1
n1::class1::memberFunc(bool b)	1	2	1
func2()	16	18	0
func3(int, long, bool, int, int, long, short)	1	0	7

Figure C.6.: Logical view - Shows the logical representation of the source code in a tree structure. The logical elements like namespaces or classes are decoupled from the physical location and merged together.



Scope	McCabe	LSLOC	NbParams
main()	1	3	0
func1(int i)	1	2	1
n1::class1::memberFunc(bool b)	1	2	1
func2()	16	18	0
func3(int, long, bool, int, int, long, short)	1	0	7

Figure C.7.: Filter view - Shows a filtered representation of the hybrid view. Possible filters are files, namespaces, types and functions.

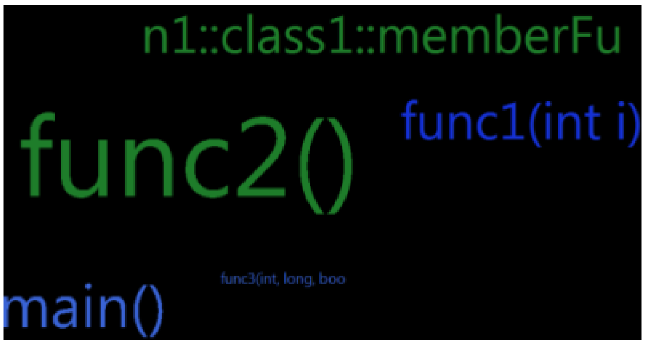
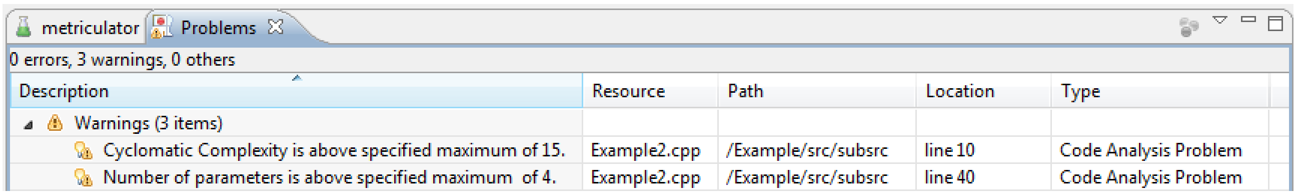


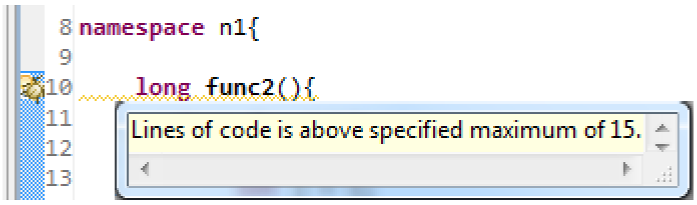
Figure C.8.: Tag cloud - visualisation of the nodes for a specific metric. The higher the metric value the bigger the font size of the name in the cloud. This tag cloud was generated for the metric LSLOC.



0 errors, 3 warnings, 0 others

Description	Resource	Path	Location	Type
Warnings (3 items)				
Cyclomatic Complexity is above specified maximum of 15.	Example2.cpp	/Example/src/subsrc	line 10	Code Analysis Problem
Number of parameters is above specified maximum of 4.	Example2.cpp	/Example/src/subsrc	line 40	Code Analysis Problem

Figure C.9.: Problems View



```
namespace n1{

    long func2(){
```

Lines of code is above specified maximum of 15.

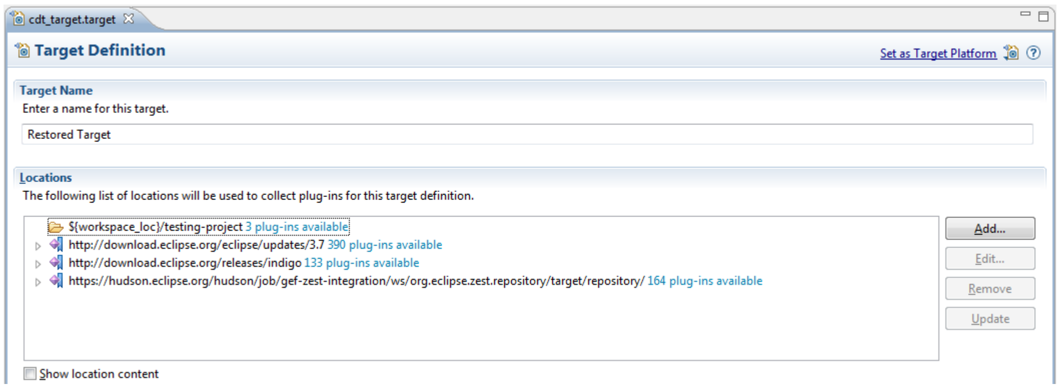
Figure C.10.: Problem marker in the source code editor with a message explaining the problem.

D. Developer Manual

Metriculator allows you to add new C++ metrics easily. There are already some metrics implemented which might help you to implement additional metrics. The following steps describe how you can add new metrics fast an simple. This manual assumes that you are working with Eclipse and have installed the Plug-in Development Environment (PDE) as well as the C/C++ Development Tooling (CDT) plug-in.

D.1. Set up

1. Checkout sources from the Git repository at sinv-56013.edu.hsr.ch/var/gitrepo/metricular.git.
2. In Eclipse *Import Existing Projects into Workspace*, select the repository checked out from point 1.
3. Set the missing baselines to *ignore* in *Eclipse > Window > Preferences > Plug-in Development > API Baselines*
4. Open the target file in the package metriculator and set it as target platform (illustrated in Figure D.1)
5. Update all the locations (illustrated in Figure D.1)



Target Name
Enter a name for this target.
Restored Target

Locations
The following list of locations will be used to collect plug-ins for this target definition.

${workspace_loc}/testing-project 3 plug-ins available
http://download.eclipse.org/eclipse/updates/3.7 390 plug-ins available
http://download.eclipse.org/releases/indigo 133 plug-ins available
https://hudson.eclipse.org/hudson/job/gef-zest-integration/ws/org.eclipse.zest.repository/target/repository/ 164 plug-ins available

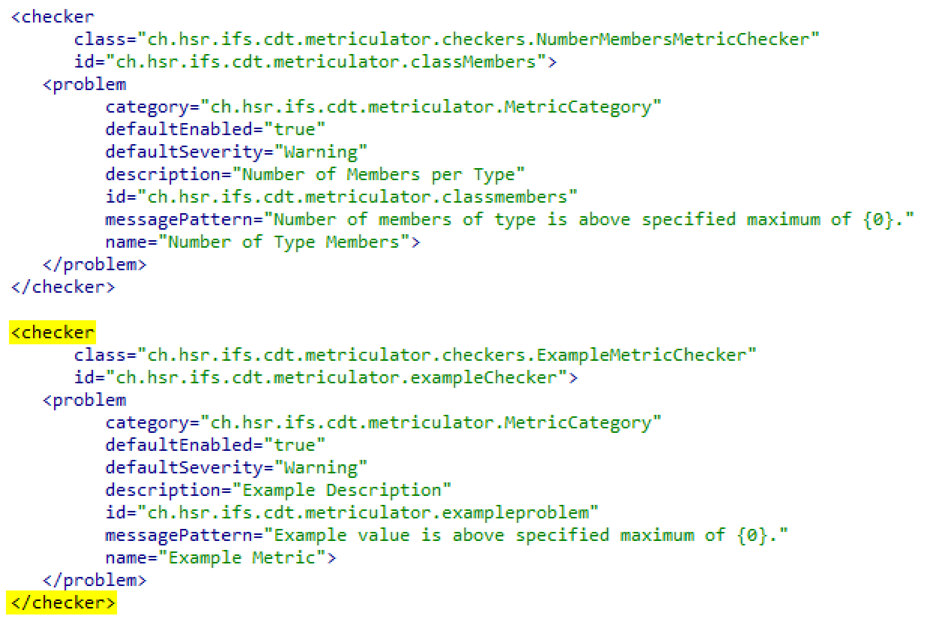
☐ Show location content

Figure D.1.: Target file

6. Clean all projects if there are still errors

D.2. Adding a new Metric

1. Add a new checker with a problem in the plugin.xml. (illustrated in Figure D.2)



```
<checker
      class="ch.hsr.ifs.cdt.metriculator.checkers.NumberMembersMetricChecker"
      id="ch.hsr.ifs.cdt.metriculator.classMembers">
   <problem
         category="ch.hsr.ifs.cdt.metriculator.MetricCategory"
         defaultEnabled="true"
         defaultSeverity="Warning"
         description="Number of Members per Type"
         id="ch.hsr.ifs.cdt.metriculator.classmembers"
         messagePattern="Number of members of type is above specified maximum of {0}."
         name="Number of Type Members">
   </problem>
</checker>

<checker
      class="ch.hsr.ifs.cdt.metriculator.checkers.ExampleMetricChecker"
      id="ch.hsr.ifs.cdt.metriculator.exampleChecker">
   <problem
         category="ch.hsr.ifs.cdt.metriculator.MetricCategory"
         defaultEnabled="true"
         defaultSeverity="Warning"
         description="Example Description"
         id="ch.hsr.ifs.cdt.metriculator.exampleproblem"
         messagePattern="Example value is above specified maximum of {0}."
         name="Example Metric">
   </problem>
</checker>
```

Figure D.2.: Part of the plugin.xml file. The highlighted element is the newly added checker.

2. Create a new metric class which inherits from *AbstractMetric* class.
   a) If your new metric requires a non default metric value aggregation override the *aggregate* method
3. Create a new metric checker class that inherits from *AbstractMetricChecker*.
   a) Define a problem ID.
   b) Create the name, description and preferences strings.
   c) Add the name, description and preferences strings to the *MetricLabels.properties* file in the package resources.
   d) Add the name, description and preferences strings to the *MetricLabels* class.
   e) Create a new instance of the metric in this checker.
   f) Register the new metric at the *MetriculatorPluginActivator* singleton instance.
   g) Implement the *reportProblemsFor* method.
   h) Implement the *processTranslationUnit* method. See chapter 3 for further information about visitors.

4. Create a new *ScopedAstVisitor*.
   a) Define the *key* string for this metric.
   b) Override the *shouldVisitxxx* fields to visit the desired AST nodes.
   c) Override and implement the *visit* method for the desired AST nodes to be visited (could be one or more visit methods).

D.3. Writing Checker Tests

Each checker has its own test class. When implementing a new checker test you should be aware of the following:

- Add your test class to the test suite by modifying AllTests.java
- You can use the *TreePrinter* class to print a tree structure to the console
- Use the *loadCodeAndRun(getAboveComment())* methods provided by Codan, to define the C++ code your checker under test will use.
- Reset the model after each test method has been ran.

E. Project Management

This chapter provides an overview to project management related tasks such as project planning and spent time analysis.

E.1. Project Plan

The project lasts fourteen weeks from September 19. to December 23. 2011. The first version of the project plan E.1 was created in week one. During the first seven weeks the initial project plan has experienced a few minor changes E.2.

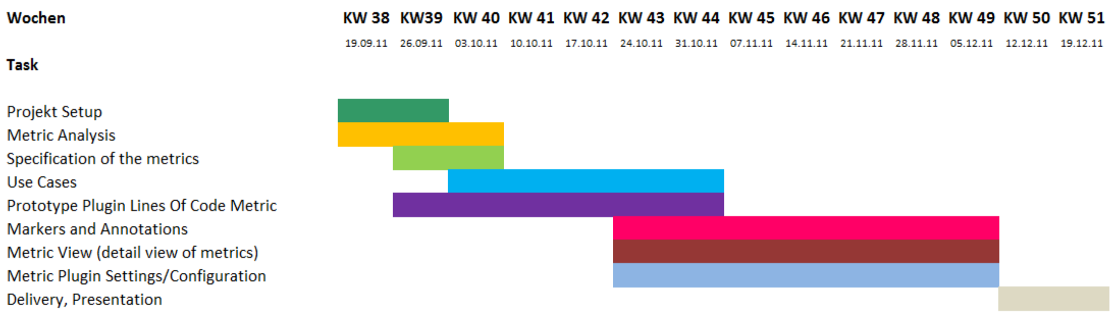


Figure E.1.: Initial version of the project plan.

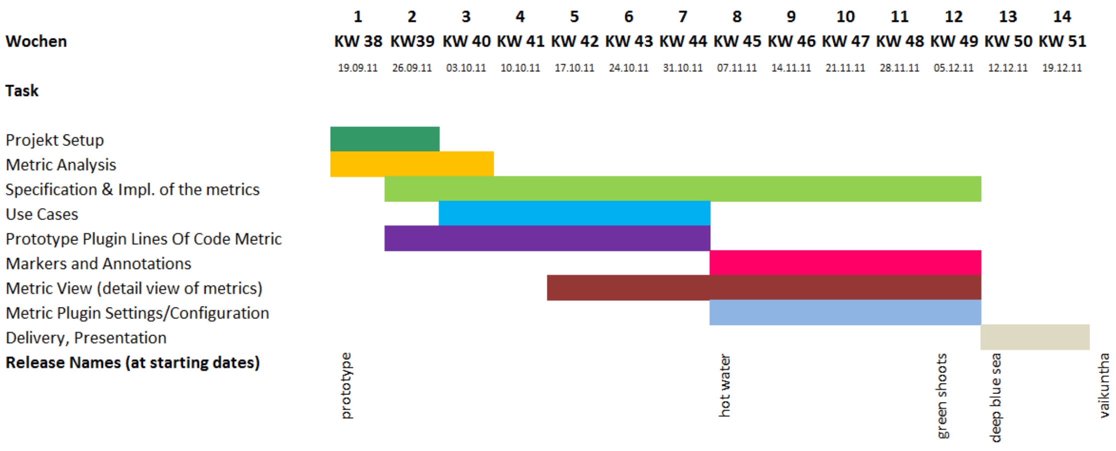


Figure E.2.: Final version of the project plan.

The task `Specification of metrics` has been renamed to `Specification & Impl. of metrics`. As we started with the implementation of the first metric (3.1) we noticed that there was no task on the project plan for metric implementation. We did not create a new one instead we extended the existing specification task since specification and implementation can not strictly be separated especially because we follow iterative development cycles.
The second change is related to user interface tasks. They were initially planned to start in week 43, two weeks before the release of the prototype. In week 41 we gladly realized that we are on schedule and that it would be reasonable to start with the implementation of the view one week earlier as planed. As we started with the view we decided to shift the markers and settings tasks to the end of the prototype phase because they were relatively easy to implement since the Codan framework provides a simple API to handle them.

E.2. Time Schedules

This chapter evaluates the time spent during the project. The first section describes the time spent per project member and week, the next section the time spent on each reported bug.

E.2.1. Spent Time per Project Member

Figure E.3 shows the time spent per member. The semester thesis module is worth 8 ECTS. This means that the expected work per week of an average student to pass the module is about 17 hours[1]. In average each of us worked about 295 hours in total, which is 57 hours (24%) above the expected 238 hours.

E.2.2. Mean Time to Fix

The mean time to fix Figure E.4 shows how long it took to fix a bug after it was reported.

The second bug has the biggest MTTF. This bug was reported because the Jenkins server was not accessible from outside the HSR LAN. Since this circumstance was not critical for us to go on working, we did not prioritize it. All other bugs were fixed within maximum nine days.

E.3. Personal Impression

From the very beginning of this project we both worked consequent and targeted to create a highly useful and simple to understand plug-in. In the sub chapters of this section each team member writes about his personal impressions during this project.
But first of all we would like to thank our advisor, Prof. Peter Sommerlad, for his

[1]8 ECTS * 30 hours per ECTS / 14 weeks

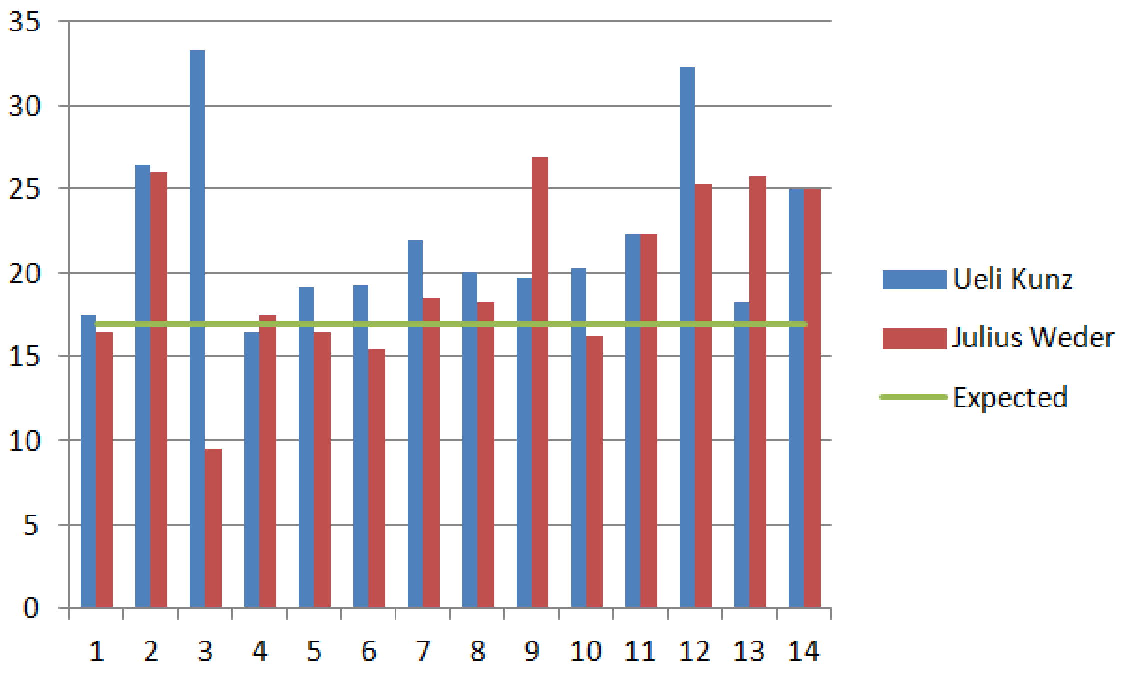


Figure E.3.: Time spent per member working on the project.

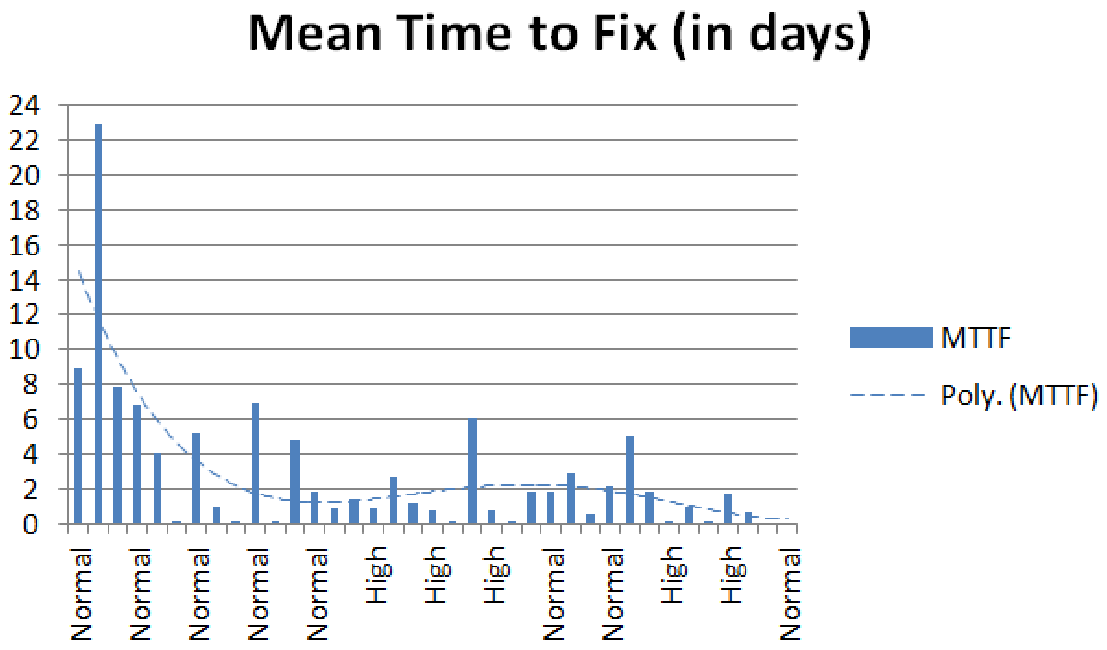


Figure E.4.: Number of days it took to fix bugs after they were reported. X axis is sorted ascending by date.

valuable time and competent advices. Special thanks goes to Thomas Corbat, who was always ready to generously assist us at technical problems.

E.3.1. Ueli Kunz

At the start of this project I was very excited what will be the outcome. I was highly motivated to deliver a well designed and stable plug-in. Although it was a small team, I was also interested in gathering more project management experiences and applying the lessons learned from earlier term projects. Getting to known with a new framework and contributing to it was also a motivating task. Since I have already benefited from other open source projects I have been willing to give something back.

At the beginning we had some extra effort setting up the project environment software. Setting up Jenkins with Git and Redmine was worth the experience but I think next time it will not bother us that much any more. To set up build automation I had to learn Maven from scratch. Afterwards I do not regret it. All in all, I feel like we spent too much time working on tasks not directly related to the goals of this thesis.

I am proud of the overall result. We reached our objectives and the plug-in is good enough to be used in medium sized projects. It has a solid design and is easily extensible.

I always had fun with Julius Weder, also during the effective extreme programming sessions late in the evenings. We often discussed and solved problems together, that was a great experience too.

E.3.2. Julius Weder

This was the biggest project in which I have participated. I learned a lot about working with the Eclipse Plug-in Development Environment (PDE) as well as Eclipse CDT and creating a useful plug-in. I could improve my programming skills extremely not only in Java and C++ but also in applying all the software engineering aspects I have learned theoretical the last years.

It was also a great experience to manage this project in reference to project management, continuous integration, unit testing, documentation and deployment. At the beginning there were a lot things to set up to finally start with core work of this project but I can now say that it will be very helpful in future. During the project I always felt highly motivated to improve the plug-in as well as my skills.

Overall, I am proud of the work we have done and it was surely a great experience which improved my professional skills and helps for future projects. At this point I would also like to thank Ueli Kunz for that he is a very competent and helpful project partner. It was during the hole project a good time as well as a intensive but productive teamwork.

List of Figures

List of Tables

F. Nomenclature

AST Abstract Syntax Tree – An abstract representation of a program or source code, usually focusing on domain specific information.

VCS Version Control System – A software that helps managing multiple versions of files.

OSGi Specification for Java runtime service and modularisation platform.

PDE The Eclipse Plug-in Development Environment provides utilities to create, maintain, test and build Eclipse artefacts.

p2 Stands fro provisioning platform and is the engine used to install plug-ins and manage dependencies in Eclipse.

Bibliography

[CDT11] Eclipse cdt project homepage. `http://eclipse.org/cdt/`, 2011.

[cla11] Project homepage of the clang project. `http://clang.llvm.org/`, 2011.

[cod11] Codan is a lightweight code analysis framework for the eclipse cdt platform. `http://wiki.eclipse.org/CDT/designs/StaticAnalysis`, 2011.

[cpp11] Iso/iec 14882. PDF, 2011.

[ecl11a] Eclipse bug report: Problems view updating too slow. `https://bugs.eclipse.org/bugs/show_bug.cgi?id=349869`, 2011.

[ecl11b] jprofiler product page. profiling tool for the java virtual machine., 2011.

[emm11] Eclemma homepage. java code coverege tool for eclipse. `http://www.eclemma.org/`, 2011.

[HF10] Jez Humble and David Farley. *Continuous Delivery - Reliable Software Releases Through Build, Test, and Deployment Automation.* Addison-Wesley, München, 1. aufl. edition, 2010.

[his11] A brief history of static analysis. `http://www.coverity.com/library/pdf/Coverity_White_Paper-SAT-Next_Generation_Static_Analysis.pdf`, 2011. see p. 2, chapter 'A Brief History of Static Analysis'.

[jen11] Jenkins set up. `https://wiki.jenkins-ci.org/display/JENKINS/Installing+Jenkins+on+Ubuntu`, 2011.

[jia11] Java ist auch eine insel - explanation of the equals method, in german. `http://openbook.galileocomputing.de/javainsel/javainsel_03_007.html#dodtp60b4b83b-b280-4e66-950f-1adf13899e67`, 2011.

[jpr11] jprofiler product page. profiling tool for the java virtual machine. `http://www.ej-technologies.com/products/jprofiler/overview.html`, 2011.

[llv11] Project homepage of the clang project. `http://llvm.org`, 2011.

[lsl11] Lsloc counting standard. `http://sunset.usc.edu/research/CODECOUNT/`, 2011.

[mav11a] Maven download. `http://maven.apache.org/download.html`, 2011.

[mav11b] Maven set up on ubuntu. http://lukieb.wordpress.com/2011/02/15/installing-maven-3-on-ubuntu-10-04-lts-server/, 2011.

[mcc11] Mccabe counting standard. http://www.verifysoft.com/de_cmtpp_mscoder.pdf, 2011. see p. 40, chapter 'Die zyklomatische Komplexität von McCabe'.

[met11a] Java metrics plug-in for ecplise. http://metrics.sourceforge.net/, 2011.

[met11b] list of metric tools. http://testingfaqs.org/t-static.html, 2011.

[mys11] Mysql password reset. https://help.ubuntu.com/community/MysqlPasswordReset#Another_way.2C_purge, 2011.

[PDE11] Official help documentation for the eclipse plug-in development environment. http://help.eclipse.org/galileo/index.jsp?topic=/org.eclipse.pde.doc.user/guide/intro/pde_overview.htm, 2011.

[red11] Redmine set up using mod passenger. http://www.redmine.org/projects/redmine/wiki/HowTo_Install_Redmine_in_Ubuntu#Ubuntu-1004-and-10041-using-Passenger, 2011.

[sou11] Sourcecloud plug-in for eclipse. https://github.com/misto/Sourcecloud, 2011.

[swt11] Article about subclassing in swt. http://www.eclipse.org/swt/faq.php#subclassing, 2011.

[upd11] metriculator p2 update site. http://sinv-56013.edu.hsr.ch/updatesite, 2011.

[Vog11] Lars Vogel. Comprehensive tutorials for eclipse developers. http://www.vogella.de, 2011.

[wik11] wikipedia article about software metrics. http://en.wikipedia.org/wiki/Software_metric, 2011.

[win11] Git set up on windows. http://help.github.com/win-set-up-git/, 2011.

The versions of the documents, referenced to in this bibliography, that we used are stored in our VCS.