Quality measurement of programming examples. Extension for programming editor BlueJ

Janusz Polok

October 3, 2008
Master's Thesis in Computing Science, 30 ECTS credits
Supervisor at CS-UmU: Jürgen Börstler
Examiner: Per Lindström

Umeå University
Department of Computing Science
SE-901 87 UMEÅ
SWEDEN

Abstract

The software measurement is a common differentiator in the software development process. It has a very big influence on project success or failure. Many software scientists proposed sets of software metrics. Using software metrics should in early project state eliminate most errors and improve the product.

However, software measurement can be also applied in the teaching process. Every teacher of programming must choose examples as a programming pattern. There are many programing handbooks and also many programming examples. But how to choose the right ones?

In this paper I will propose a set of software metrics for evaluation of programming examples and will present the SMT tool, an extension for the programming editor BlueJ.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

For the last couple decades software quality changed in many levels. Computer programs have come a long way from punched cards to window oriented applications. As users we can see the obvious difference between text programs from the late 80's and today's complex systems. This continued progress has resulted in software quality becoming more of a differentiator between products. Today's developers need to be aware of not only about a look and stability but also about such things as readability, comprehension and complexity of written software. Object-oriented software architecture give programmers large scope to develop software. This leads us to the question: "How to measure software quality?".

This part of software engineering is still open. Many people have developed different types of measuring techniques; some are useful, some not necessarily so. Some of the developed metrics are based on code analysis and some on class diagrams, that is only for object-oriented software. In the object-oriented approach of software projecting, when it is known what should be done - all requirements are known - developers start from class diagrams. For this stage a special language was invented called UML (Unified Modeling Language) [19].

Many software metrics were developed to measure the quality of software based on class diagrams [14]. This approach can allow software evaluation at a very early stage.

The next stage in system development process is coding. How to write code is a lesson that every computer scientist must pass. All must learn a group of commands and a number of compiler instructions with which to write their first program. The fastest way to learn is by following examples from programming handbooks. In the very beginning we do not care if our program looks nice, or if our classmates can understand our approach in problem solving. The most important criterion is that it works. But after a while the young programmer will begin to attempt programming of much greater complexity. Somewhere in between this a programmer needs to learn how to write software that is understandable for others. Object-oriented languages give classes, which can be reused in other applications. To do so the class needs to be well written and easy to further develop.

As I mentioned before, the main source of knowledge is examples. From them programmers learn how to build their software and how to write it in a readable way.

Chapter 2

Problem Description

2.1 Problem Statement

How can young programmers know if their piece of code is well written when all knowledge about programming comes from lectures and example programs? Many examples show how to use the same components or methods but how can we choose examples that are more digestible for students? Teachers have a huge responsibility on their shoulders. They must choose the right example programs for their students. Students put full trust in materials given by teachers. Examples given by teachers are a base source of knowledge about how the written code should look and how the source code should look to be more readable to others. Readability of the source code is one of the most important metrics. However, in the Java language information on how to write source code was presented by Sun Microsystems in Code Conventions for the Java^{TM} Programming Language [21] and almost everyone follows these rules. This is why my thesis will not analyze readability of source code. I worked with the assumption that every example is written consistently with Code Conventions for the Java^{TM} Programming Language.

My thesis will therefore be focused on the complexity and comprehension metrics of source code.

2.2 Goals

How can a young programmer check in an easy way which example is worth following and which will not be worth an effort of exploring? In my thesis I will propose a set of metrics regarding software quality based on source code analysis. As a measuring tool I will present an extension for the Java programming editor for beginners BlueJ [24].

2.3 Related Work

Software measurement is already a part of many software development kits (SDK). A very good example is the Eclipse development environment with the Metric Plugin for Eclipse [7]. In the paper "Advanced Object-Oriented design, Object-Oriented metrics" [18] two students describe the evaluation of Jakarta Common HttpClient ver.3.1 [3]. In report is deep analysis based on Chidamber and Kemerer metrics (CK metrics, see section 3.1.2) [6].

There are tools calculating software metrics based on class diagrams. An excellent example is OMetrics, described in "Metrics MOOD in Rational Rose" [10]. This tool calculates CK and MOOD metrics based on data from class diagrams in a project in Rose.

It is also possible to evaluate source code with Imagix 4D [11], which generates a series of over 70 source metrics. But then the user must analyze the result and for programming examples analyzing 70 different metrics is a bit too much.

On the BlueJ web site [24] a set of extensions for BlueJ can be found, which evaluate source code. For example CEB extension. It makes evaluation based on CK metrics set. There is no available extension for BlueJ with wide metrics set and possibility to configure counting rules. That is why I made SMT (Software Measure Tool), an extension for BlueJ. In the SMT the user has influence on counting rules for Halstead's metrics, can choose metrics to analyze and results are presented in a commented table. Results can be also saved in a file.

2.4 Organization of the thesis

First in Chapter 3 I will present the "most popular" software metrics. Than I will say a few words about software measurement in the teaching process and features of good examples. Next, in Chapter 4, I will bring closer the set of metrics that, in my opinion, shows all important features of examples. Then I will show the way the work was done. In Chapter 6 I will propose a results interpretation. And in Chapter 7 I will present conclusions and suggestions for future work.

Chapter 3

Software measurement

In this chapter I will present the set of "most popular" metrics. Please, take under consideration that this is an academic project and some of them will be very cursory described. However, metrics important for a project I describe with more details. Next I will talk about software measurement in teaching and learning process to conclude with attributes of good examples.

What measurement is everyone knows. We measure almost everything. Distances, budgets, combustion of our cars. Some people measure distance from work or school in steps. According to Norman E. Fenton "measurement is the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules" [8]. In the case of software measurement an entity is a piece of software, and attributes are features or properties of the entity. Attributes and properties can be described with software metrics.

3.1 Examples of software metrics

In the software development process software metrics are commonly acceptable and used. There is a specific group of object-oriented software metrics. Some of them are adjusted to specific a programming language, and some are language independent. A very good overview software metrics is available in Bluemke's article "Metrics for Assessing Complexity and Testability of Object-Oriented Software" [5].

Metrics can be used to evaluate a single class or a whole project. I will give examples of software metrics mainly based on the "A survey of metrics for UML class diagrams" [14] report.

3.1.1 Lines Of Code

This metric is widely used in almost every evaluation tool. Some researchers count a line that is not a commented line and not an empty line as a code line. Others count just lines that are "real" code. That is lines containing an instruction. Lines with only "{" or "}" will not be counted as code lines.

LOC, independent from counting rules, shows the physical size of source code.

3.1.2 Metrics for Object-Oriented Design (MOOD)

Abreu Brito F.,Goualo M., Esteves R J. proposed a set of six metrics for Object-Oriented projects [2] :

- The Method Hiding Factor (MHF)
- Attribute Hiding Factor (AHF)
- Method Inheritance Factor (MIF)
- Attribute Inheritance Factor (AIF)
- Coupling Factor (CF)
- Polymorphism Factor (PF)

The MHF and AHF metrics show how secured the software is, how good information is hidden. They show how many classes (MHF) and attributes (AHF) are declared as private.

The MIF, AIF and PF metrics show Object-Oriented (OO) design mechanisms: inheritance and polymorphism.

The CF metric shows the coupling factor between classes excluding coupling through inheritance.

Ilona Bluemke and Piotr Zajac give an excellent example of how these metrics can be implemented and how they should be interpreted [10].

MOOD metrics are defined by following equations:

The Method Hiding Factor (MHF)

$$MHF = \frac{\sum_{i=1}^{TC} \sum_{m=1}^{M_d(C_i)} (1 - V(M_{mi}))}{\sum_{i=1}^{TC} M_d(C_i)}$$

where

$$V(M_m i) = \frac{\sum_{j=1}^{TC} is\_visible(M_{mi}, C_j)}{TC - 1}$$

$$is\_visible(M_{mi}, C_j) = f_0^{1 \Leftrightarrow J \neq i \wedge C_j \ may\ call\ M_{mi}}{}_{otherwise}$$

TC = total number of classes in the system under consideration
$M_d(C_i) = M_v(C_i) + M_h(C_i)$ = methods defined in the class C_i
$M_v(C_i)$ = visible methods in class C_i (public methods)
$M_h(C_i)$ = hidden methods in class C_i (private methods)

Attribute Hiding Factor (AHF)

$$AHF = \frac{\sum_{i=1}^{TC} \sum_{m=1}^{A_d(C_i)} (1 - V(A_{mi}))}{\sum_{i=1}^{TC} A_d(C_i)}$$

where

$$V(A_m i) = \frac{\sum_{j=1}^{TC} is\_visible(A_{mi}, C_j)}{TC - 1}$$

$$is\_visible(A_{mi}, C_j) = f_{0 \quad otherwise}^{1 \Leftrightarrow J \neq i \wedge C_j \ may\ call\ A_{mi}}$$

$A_d(C_i) = A_v(C_i) + A_h(C_i)$ = attributes defined in the class C_i
$A_v(C_i)$ = visible attributes in class C_i (public attributes)
$A_h(C_i)$ = hidden attributes in class C_i (private attributes)

Method Inheritance Factor (MIF)

$$MIF = \frac{\sum_{i=1}^{TC}(M_i(C_i))}{\sum_{i=1}^{TC} M_a(C_i)}$$

where
$M_a(C_i) = M_d(C_i) + M_i(C_i)$ = available methods in class C_i
$M_d(C_i) = M_n(C_i) + M_o(C_i)$ = methods defined in class C_i
$M_n(C_i)$ = new methods in class C_i
$M_o(C_i)$ = overriding methods in class C_i
$M_i(C_i)$ = inherited methods in class C_i

Attribute Inheritance Factor (AIF)

$$AIF = \frac{\sum_{i=1}^{TC} A_i(C_i)}{\sum_{i=1}^{TC} A_a(C_i)}$$

where
$A_a(C_i) = A_d(C_i) + A_i(C_i)$ = available attributes in class C_i
$A_d(C_i) = A_n(C_i) + A_o(C_i)$ = attributes defined in class C_i
$A_n(C_i)$ = new attributes in class C_i
$A_o(C_i)$ = overriding attributes in class C_i
$A_i(C_i)$ = inherited attributes in class C_i

Coupling Factor (CF)

$$CF = \frac{\sum_{i=1}^{TC}\left[\sum_{j=1}^{TC} is\_client(C_i,\ C_j)\right]}{TC^2 - TC}$$

where
$is\_client(C_i, \quad C_j) \quad = \quad f$
$_{1 \Rightarrow C_c \Rightarrow C_s \wedge C_c \neq C_s}$
$_{0 \quad otherwise}$
$C_c \Rightarrow C_s$ represents relation between C_c and serving class C_s

Polymorphism Factor (PF)

$$PF = \frac{\sum_{i=1}^{TC} M_o(C_i)}{\sum_{i=1}^{TC} [M_n(C_i)\ \times\ DC(C_i))]}$$

where
$M_o(C_i)$ = overriding methods in class C_i
$M_n(C_i)$ = new methods in class C_i
$DC(C_i)$ = number of descendants of class C_i

3.1.3 Chidamber and Kemerer metrics (CK metrics)

Chidamber and Kemerer proposed a set of six metrics [6]:

- Weighted Methods per Class (WMC)
- Depth of Inheritance Tree (DIT)
- Number Of Children (NOC)
- Coupling Between Objects (CBO)
- Response For a Class (RFC)
- Lack of Cohesion in Methods (LCOM)

CK metrics were defined to evaluate single classes or small groups of classes, not for whole systems. That is why CK metrics are used mainly by programmers and hardly by any system designer.

In following sections I will briefly describe CK metrics.

Weighted Methods per Class

Chidamber and Kemerer define this metric in following way [6]:

Consider a class C1, with methods M_1,..., M_n that are defined in the class. Let $c_1, ..., c_n$ be the complexity of the methods. Then:

$$WMC = \sum_{i=1}^{n} c_i$$

I simplified this metric a bit and assumed that the methods' complexity is an unity. This reduces the WMC to number of methods in class.

Depth of Inheritance Tree

This metric measure how many ancestor classes can potentially affect the considered class. In the case when we use Java programming language DIT can equal zero or one. Zero in case when the class does not inherit from any other class, and one when it does. In Java multiple inheritance is not allowed and DIT can not exceed one.

Number Of Children (NOC)

Number Of Children is the number of immediate subclasses subordinated to a class in the class hierarchy.

Coupling Between Objects (CBO)

Metric is a level of dependency between classes. Inheritance dependency is not taken under consideration.

Response For a Class

Chidamber and Kemerer define this metric in the following way [6]:

$$RFC = |RS|$$

where
RS is the response set for the class.
The response set for the class can be expressed as

$$RS = \{M\}U_{alli}\{R_i\}$$

where
$\{R_i\}$ = set of methods called by method i
{M} = set of all methods in the class.

The response set of a class is a set of methods that can potentially be executed in response to a message received by an object of that class. The cardinality of this set is a measure of the attributes of objects in the class. Since it specifically includes methods called from outside the class, it is also a measure of the potential communication between the class and the other classes.

I define a RFC as a number of methods called by any method of this same class. This also includes recursions. The defined metric tells us how many methods are never used by sibling methods and are used only as a link by other objects. Usually sibling methods do not use any "get...", "set..." or "is..." method. Instead they recall directly to the instance's attributes. That is why the RFC is usually the WMC reduced by getting, setting and checking methods.

Lack of Cohesion in Methods (LCOM)

The metric is a cohesion level between a class' methods. Attributes and their usage are compared. The LCOM is a count of the number of method pairs, whose similarity is 0 minus the count of method pairs whose similarity is not 0 [6].

3.1.4 Cognitive Functional Size

The Cognitive Functional Size (CFS) metric developed by Jingqui Shao and Yingxu Wang and presented in April 2003 [20] is a software complexity measure based on cognitive weights.

When we want to talk about CFS the indissoluble part of it is the cognitive weight (C_w) of software [20]. Shao and Wang describe cognitive weight as a degree of difficulty or relative time effort required to fully comprehend an analyzed piece of software. Each piece of software analysis begins with the architecture, then we look deeper and begin to analyze the basic control structures (BCSs) of the software. BCSs are a set of instructions that are used for building software logic. In my implementation of CFS I distinguish the following BCSs:

- embedded component - method calls from current class and recursions
- sequence - at least one instruction that do not match to above ones and is not commented

Category	BCS	W_i
Sequence	sequence	1
Branch	if-[else]	2
	switch-[case...]	3
Iteration	for	3
	while	3
	do (while)	3
Embedded component	method call	2
	recursion	3

Table 3.1: Cognitive weights

The cognitive weights used by me are shown in Table 3.1.

However, sequence is counted if and only if statements are written in the main command branch, e.g.

the code:

```
stmt;
...
stmt;
if () {...}
stmt;
...
stmt;
```

will have a $C_w = 4$
sequence+branch+sequence (1 + 2 + 1 = 4)

and the code:

```
stmt;
...
stmt;
if () {
   ...
}else {
 stmt;
 ...
 stmt;
}
```

will have a $C_w = 3$
sequence+branch (1 + 2 = 3)

As we can observe, the last sequence is locked up in the else clause and is not counted.

Shao and Wang write about two different architectures for calculating cognitive weights (W_i). They propose that the total cognitive weight of a software component, W_c, is defined as the sum of the cognitive weights of its q linear blocks composed of individual BCSs. If blocks are embedded one in another then cognitive weight of external block is multiply with weights of inner BCSs. The general formula looks as follows:

$$W_c = \sum_{j=1}^{q}[\prod_{k=1}^{m}\sum_{i=1}^{n} W_c(j,k,i)]$$

where

– W_c - cognitive weight

– q - number of linear external blocks

– m - number of layers in nested BCSs

– n - number of linear blocks in k-layer

The cognitive functional size of software is calculated from equation:

$$S_f = N_{I/O} * W_c$$

where

– S_f - cognitive functional size

– $N_{I/O}$ - sum of inputs and outputs

– W_c - total cognitive weight

S_f is equal to 0 in cases where:

– sum of input and outputs is equal to zero, e.g. class' constructor without arguments

– total cognitive weight is equal to zero, e.g. method without any commands

– both are equal to zero

By definition the class or method does not take any effort from us to understand it. However, when the method is a "void" type method and does not take any arguments its S_f is zero; we lose all information about the method's cognitive weight. So this metric can turn into wrong conclusions. It suggests that this method is extremely easy to grasp. The cognitive weight of that method does not need to be equal to zero, it can be extremely difficult to comprehend (very complicated) method with high C_w. In order to avoid these situations I decided to change counting rule with one condition: When the sum of inputs and outputs equals zero then $S_f = W_c$ ($N_{I/O} = 1$). This condition will preserve important data about the complexity of "void sampleFunction(void)" functions.

3.1.5 Cyclomatic complexity

The cyclomatic complexity (CC), first introduced by McCabe [15], shows maximum number of linearly independent circuits.

The cyclomatic complexity shows how complex designed software is. It literally shows the number of possible ways how the program can be executed. Originally CC was developed as a graph-theoretic software complexity measure. McCabe [15] describes how CC is calculated based on graph decomposition.

A program written as a set of sequential commands without any conditional instructions has cyclomatic complexity equal to 1, because program can be executed only in one

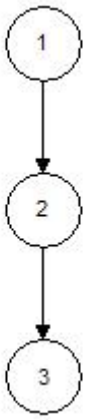


Figure 3.1: Program with $CC = 1$.

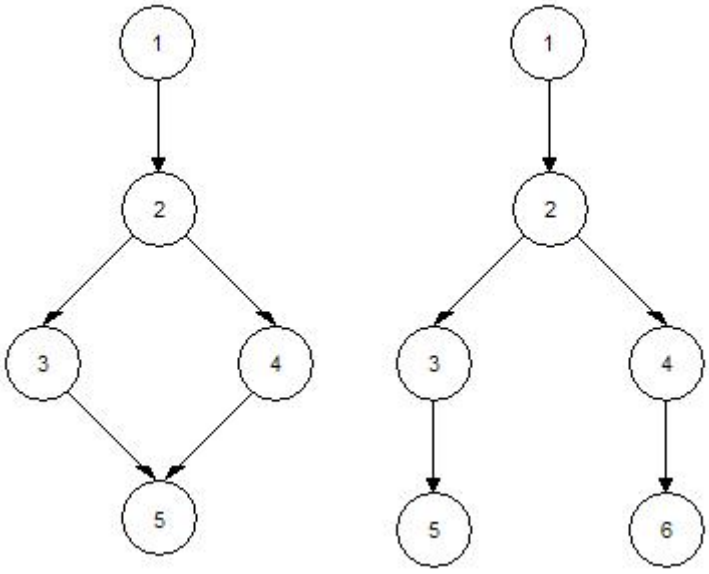


Figure 3.2: Programs with $CC = 2$.

way (Figure 3.1). When a code contains some conditional commands the program gets more complicated and execution of this code can proceed in more than one way. When the code consists of sequential commands and one branching instruction, for instance an if()[else] statement (Figure 3.2), it can be executed in two ways. One way is when the condition is fulfilled (node 3 is visited) and the second when the condition has failed (node 4 is visited). As we can see on Figure 3.2 a program with CC=2 can be built in a couple ways.

To determine the cyclomatic complexity of given statement we need to designate how many different ways it produces. But there is an exception. When we want to analyze the if()[else] statement it always produces two ways (Figure 3.2). However, in case of multi-conditions in one if()[else] statement we determine CC in a different way:

```
code:
      stmt;
      ...
      stmt;
      if ( someVaraible == 0 ) {
        stmt;
        ...
        stmt;
      }else {
       stmt;
       ...
       stmt;
      }
```

This pseudo-code is represented by Figure 3.2 (right) and its cyclomatic complexity is 1, because it adds only one alternative way in program's flow possibilities. However, the if()[else] statement can contain a couple conditions, and if this is the case the complexity of that statement increases. In some cases many programmers, who are not comfortable with short switch()[case] statement use multi-conditional if()[else] statements. In these cases the cyclomatic complexity if()[else] statement is equal to the number of conditions separated by && and ||, e.g.:

```
the code:
          stmt;
          ...
          stmt;
          if ( someVariable == 0 || otherVariable == 0 ) {...}
          stmt;
          ...
          stmt;
CC = 2,   2 - conditional statement
but CC of the entire code equals 3, because we need to add
the main program flow (main program branch)

and the code:
          stmt;
          ...
          stmt;
```

```
if ( someVariable == 0 && otherVariable == 0 || someComponent.isVisible() ) {...}
stmt;
...
stmt;
```

CC = 3, 3 - conditional statement
but CC of the entire code equals 4, because we need to add
the main program flow (main program branch)

Every multi-conditional if()[else] statement can be represented as a set of single condition if()[else] statement, which all produce one additional program flow [15]. In the Java language, besides if()[else] , there is another branching instruction: switch()[case] . Following the main rule about determining CC of a statement, in case of a switch()[case] it is equal to number of cases considered by switch()[case] , excluding the "default" case.

the code:

```
stmt;
...
stmt;
switch( someVariable ) {
        case 1: ....
        case 2: ....
        default: ....
}
stmt;
...
stmt;
```

CC = 2, 2 - cases statement (Figure 3.3)
but CC of the entire code equals 3, because we need to add the main program flow (main program branch)

and the code:

```
stmt;
...
stmt;
switch( someVariable ) {
        case 1: ....
        case 2: ....
        case 3: ....
        case 4: ....
}
stmt;
...
stmt;
```

CC = 4, 4 - cases statement
but CC of the entire code equals 5, because we need to add the main program flow (main program branch)

To calculate cyclomatic complexity for a class the first thing to do is to determine the CC for each method of the class. CC for whole the class is a sum of the methods' CC. Depending how big the class can be the cyclomatic complexity can be a large number. That is why in most analysis the average cyclomatic complexity is taken ($avg(CC)$). The average cyclomatic complexity is a class' CC divided by number of methods(M):

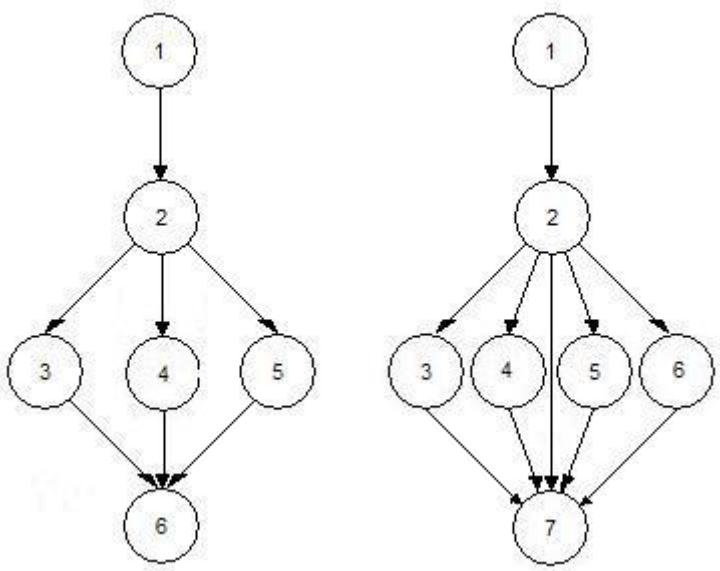


Figure 3.3: Program with switch()[case] statement

$$avg(CC) = CC/M$$

The SMT extension gives the average cyclomatic complexity (avg (CC)) metric and the biggest cyclomatic complexity (maxCC) from all the class' methods. The set of these two metrics combined with Weighted Methods per Class (WMC), from the CK metrics set, give a wide overview of class complexity.

3.1.6 Halstead's Software Science

Maurice Halstead's Software Science [9] was originally developed to measure the psychological complexity of the programming task [17]. Halstead's measures are based on physical characteristics such as the number of operands and operators, of the program's source code.

In Halstead's software science we have following concepts [9]:

- n_1 - number of unique operators
- n_2 - number of unique operands
- N_1 - total number of operators
- N_2 - total number of operands

Using these numbers we can calculate metrics:

- $n = n_1 + n_2$ - size of used vocabulary
- $N = N_1 + N_2$ - program length
- $V = N * log_2 n$ - Volume

- $L = (n_1^* * n_2)/(n_1 * N_2)$ - Level
- $V = 1/L$ - Difficulty
- $E = V/L$ - Effort

where
$n_1^* = 1$ + M, with M = number of modules
What is an operand and an operator?
As operands can be classified all variables, constants and literals:
the code :
int newVarible;
final double roundPi = 3.14;
contains two operands (newVariable and roundPi). Shortly, an operand can be described as a data object.

As an operator is classified every program keyword, including scope defining keywords such as do{}while(). The list of all operators depends on the programming language used in the implementation. In my case it is a Java language. Every language has a list of operators, which the programmer can use in his work. Java is not alone in this. That is why I used the standard Java operators list to distinguish all operators in analyzed piece of code.

The complex list of available operators can be found in Appendix C. Not every operator is taken under consideration in counting rules. However SMT gives an opportunity to customize the operators set (see User's Guide).

3.2 Software Measurement in Teaching/Learning Process

Every computer scientist have participated in at least one programming course and is able to determine which one was good and whith one was bad. In the grading process usually are taken: teacher's knowledge and quality of examples, on which teaching was based. Usually these two correlate. Many have experienced errors appearing during compiling the example programs or discover sour truth that the analyzed piece of code has nothing to do with concept that it should taught. Good example should have the following properties [13]:

- understandable by a computer
- understandable by students
- effectively communicate the concept(s) to be taught
- exemplify one, or very few, new concepts at a time
- fulfill teaching strategy

Every teacher have to make a choice of which examples use in course. There are many external factors that should be taken under consideration, such as students age, knowledge level etc. Every example should contain additional comments according new concepts to help students fully understand a problem and clear all doubts.

In my thesis I leave the comments in examples as a separate part to analyze. I do not take comments under consideration, every calculations are based on code written in the Java programing language.

Software measurement research began in the 1970's. Some researchers focused on the design level of software development. These researchers resulted in many software metrics for UML Class Diagrams [14]. Measuring class diagrams quality allows in a very early stage of the software development process to identify weak design spots and to predict maintainability and reusability.

In my thesis I focus on the next stage of the software development process: implementation. All calculations are based on information extracted from source code.

3.3 Software Measurement for Quality of Example Programs

Software as a product can be graded. The user can estimate software quality as good or bad in the same way as he can grade the quality of service given. Every field of our life is estimated within distinct rules. If we are to make a decision we need some information; we need to know all differences between products or services. When we study for some test and the lecture is not very well understood we search for other sources of knowledge. Usually we ask older friends for help to study or to choose "the right" textbook. Usually information where "you have lots of well written examples" tips the balance. The examples in every field of science are the most important source of knowledge for students.

This brings us to the question: "Which examples are good, and which are bad?". There are many opinions how to measure goodness of examples. In software engineering, researchers have mixed opinions: some of them think that software can not be analyzed by measurement, and others claim software can be measured. In past decades researchers and practitioners proposed more than one thousand software measures among over five thousand papers. However, until now no one has ever presented a set of metrics which will say that this software is good and other is bad.

Software measurement is a very wide research field. In Object-Oriented Development software measurement research is split in two major parts: UML class diagrams measurement and source code measurement. The first one is directed to early error detection and is aimed at the design level of the software development process. However, metrics designed for UML class diagrams are also used in source code analysis and vice versa.

What sort of measures should be taken under consideration in examples' analysis? The most important aspect for a young programmer is the readability of an analyzed piece of code. In order to focus on the Java programming language, I will assume that every example taken under consideration is written in accordance with Java code convention [21] and therefore readability is not an issue.

A good example must be understandable by both students and computer. For one problem there are many solutions. Many examples solve this same programming problem, but some of them are much easier to comprehend than others. I had to decide how to in a set of metrics show the amount of effort required to comprehend the example. I posed the question: "what makes code easy to understand?" Probably for everyone the answer on this question will be different.

When I started following papers in software measurement I encountered many different measures. However, too much information can cause confusion. That is why I decided to implement metrics that will describe examples from the most important parts, from the students point of view. I decided that my set of metrics should show:

- how many attributes and methods the student has to deal with
- how many possible ways the program can be executed
- how complex the written code is
- how much effort it will cost a student to get through it

I had to choose a set of metrics, which will provide this information and this was not a simple task. There is a wide spectrum of papers regarding software measurement, but as previously discussed, there have not been distinguished a set of metrics that will satisfy everyone. I propose to choose metrics that will give information most important for young programmers and their lecturers and can be easily interpreted by them.

The authors of "Toward a Measurement Framework for Quality of Example Programs" [13] give a set of properties which should be taken under consideration in analyzing an example program:

- Readability
- Structural complexity
- Cognitive complexity
- Commenting
- Size
- Consistency
- Presentation
- Progression
- Vocabulary

Chapter 4

Chosen metrics

I read many reports and I had many discussions regarding software measurement. However, none of them gave a full list of metrics that will describe a programming examples. None of the described metric, or set of many, will give a definitive answer on the question: "Is this example good or not?" That is why I made a choice based mainly on system requirements and related work.

In the decision which metrics to choose, I followed the framework described at the end of the previous chapter [13] and method from paper by Linda Westfall "12 Steps to Useful Software Metrics" [25].

According to Westfall most important is to determine who will use a developed software, and for what?

The SMT extension is aimed at Java teachers and students. BlueJ is an environment used only for basic programming education. Team's projects or bigger single projects are not usually produced in BlueJ. That is why all metrics for analyzing complex systems are in this case to overlooked. The whole MOOD metrics set is designed for complex systems. In "MOOD Metrics in Rational Rose" [10] the authors made an evaluation on four systems. With help of OMetrics [10] tool evaluation was made on class diagrams. It is an example how MOOD metrics can be used in evaluation of big systems in the project design stage. For the SMT requirements MOOD metrics are not adequate.

In my work I made the assumption that all analyzed examples are written in convention for Java code [21]. It is very difficult to automatically distinguish a "well" written from a "poorly" written example. That is why I have not implemented any metric for readability and leave this aspect for further work.

Structural complexity I decided to show with Cyclomatic Complexity (CC) [15]. This metric shows how many possible control flows are in the evaluated example. CC gives very useful information about complexity, is simple to analyze and SMT users will have no problems with results interpretation.

One of the most important metrics are metrics which show cognitive complexity of example program. Cognitive complexity measure is based on Halstead's measures or information theory. I chose to implement a set of metrics based on Halstead's science (see section 3.1.5). These metrics are a very good basis for evaluating software cognitive complexity.

To give more information about software complexity I have implemented "Cognitive Functional Size" (CFS). This metric is based on cognitive weights which describe a difficulty or relative time effort required to comprehend a software chunk.

I do not try to analyze a comment's quality or example consistency. These parts of the evaluation process are very difficult to automate. The system will have to know what kind of knowledge the student has, what kind of knowledge will be for them known, and what big part of new information are they able to learn with one example. The system will also have to know the teaching concept to make a good choice between teaching approaches. At this moment I assume that user is able to make this part of evaluation.

Comments' quality is very subjective evaluation. For one of us the same piece of software is a very well commented, and for others it has to much not necessary comments. That is why I decided to not consider comments in example evaluation. Besides, all necessary explanations should be given on a lecture.

In evaluation of programming examples I do not take under consideration the physical size of code and that is why I decided to not implement LOC metric.

Chapter 5

Accomplishment

5.1 Preliminaries

To meet the goals of my project I need to design and implement an extension for the BlueJ programming editor. This extension should evaluate software metrics for Java source code. To deal with this task I needed to:

- designate a set of metrics
- find a way to decompose text of a source code, extract useful data and store it in some easy to search and interpret form
- find a way to link up the decomposition tool with BlueJ as an extension
- implement metrics

5.2 How the work was done

The first step of each project is always research. My thesis base on software measurement and my research was directed on object-oriented software measurement. After analyzing numbers of publications in software measurement and many discussions with my tutor, the set of metrics was chosen. When the set of metrics was known I met a question: "How do I implement these metrics?".

5.2.1 Source code decomposition

The main proposal was to create an extension for the Java programming editor BlueJ. Therefore the second step was to research BlueJ and the extension possibilities. After acute analysis and help from BlueJ's developers it proved to be that BlueJ does not have any tools for source code analysis available in extension. I had to start looking for a tool in source code decomposition. At this phase I started to wonder how I should obtain data from text and how to store it. Data structures are a very wide area of computer science. Whilst I was researching useful data structures I encountered Abstract Syntax Tree (AST). As the name suggests it is a data tree where each interior node represents a programming language construct and the children of that node represent meaningful components of the construct [26]. I started looking for a text parser with a tool to

create an AST. ANTLR (Another Tool for Language Recognition) [22] also used by BlueJ developers, however it is not available in the extensions API.

ANTLR is a parser and translator generator tool that lets one define language grammars in either ANTLR syntax (which is YACC and EBNF (Extended Backus-Naur Form) like) or special AST syntax [4].

How does it work?

Text recognition breaks up an input stream of characters into tokens. Using ANTLR you need to create a grammar file. The grammar file contains subclasses of Lexer, Parser, or TreeParser. The lexer, also called scanner, lexical analyzer or tokenizer, is responsible for quantification of the input stream into discrete groups called tokens. Tokens are components of the programming language in question such as keywords, identifiers, symbols, and operators. The lexer converts the input stream of characters (source code) into a stream of tokens which have individual meaning as dictated by the lexer's rules. The parser, also called a syntactical analyzer, organizes the tokens it receives from lexer into the allowed sequences defined by the grammar of the language. In this case parser converts the sequences of tokens that it has been deliberately created to match into AST form. Example AST's are presented in Figure 5.1.

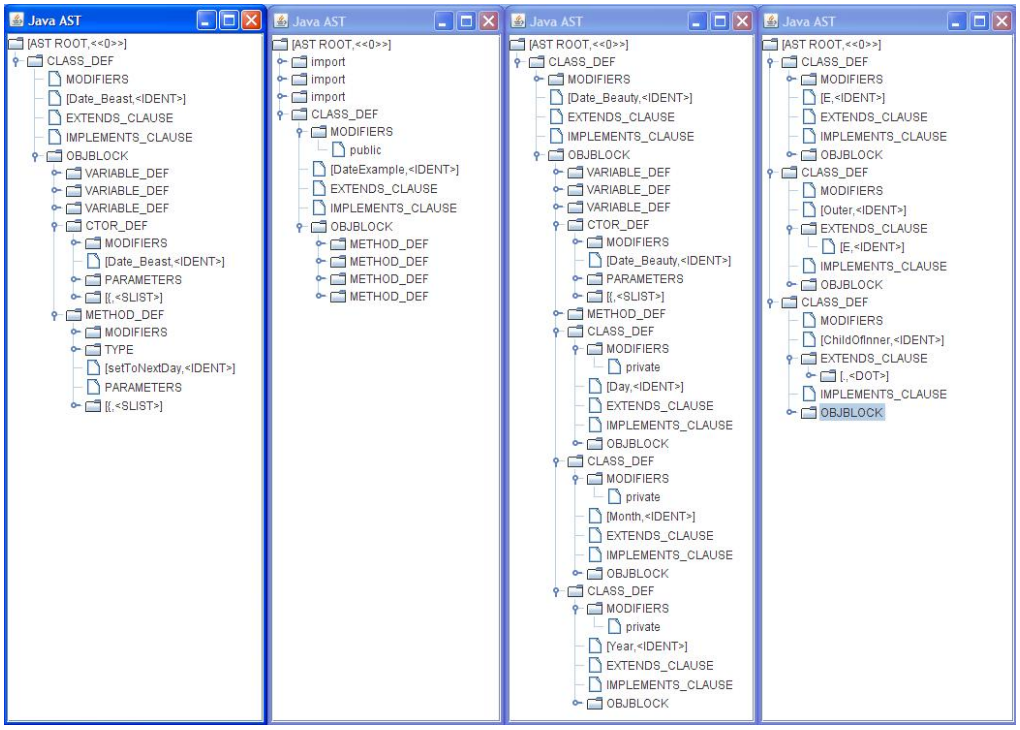


Figure 5.1: Examples of Abstract Syntax Trees

After preliminary research I found an open-source grammar for the Java language [1]. At this time there was only one stable version for Java parsing with a tree walker. Unfortunately this version is dedicated for Java version 1.3 and earlier. As a result of compiling the grammar file [1] with ANTLR we get implementations in Java programming language of lexer, parser and token types.

5.2.2 Extension for BlueJ

The BlueJ team gives a very useful introduction for writing extensions [16] supported by very wide a API [23], with help from this I started work on my extension.

The BlueJ's extensions use proxy object to get access to the BlueJ application. The main extension class inherits from the extension class. To ensure action reaction communication between BlueJ and the extension, main class should also implements the PackageListener interface. Thanks to this relation extensions can react to the user's action on classes or projects edited and objects created by BlueJ. An extension can add new menu items to the BlueJ Tools, Class and Object menus. The skeleton of the simplest extension is shown on the BlueJ web site [16]. I used this example to build my extension.

I called my extension SMT from Software Measure Tool. The main class is SMT which inherits from Extension class. The SMT class is based on SimpleExtension class [16] and represents a skeleton of the SMT extension. In this class all menus are established and actions bounded with BlueJ. I decided that class evaluation should be called after the class level and I added the class menu options. After a while I also added there an option for evaluation of the whole project. I think this is most intuitive solution. I also think that users should have some influence on software used, which is why I added an options window accessed by the Tools menu. When I had a draft of my extension and classes for text decomposition from ANTLR, I began metrics implementation.

General overview

As I mentioned before, class' evaluation is called by user's action on Class menu. The difference between Evaluate Class and Evaluate Project lies in the number of classes passed to evaluation, the evaluation process looks the same.

The list of classes for evaluation is passed as a parameter to the constructor of the new ResultsWindow class' object along with the name of a class or project. The ResultsWindow object is a window where all results are presented. The ResultsWindow class coordinates source code decomposition with metrics calculations and data representations.

On start it derives an option across SMToptionsWindow class static methods. When object arranges all data, source code decomposition stage begins:

- JavaLexer instance is created
- JavaParser instance is created with JavaLexer object as a parameter
- parsing method is called
- AST from JavaParser object is passed to Scheduler constructor
- Scheduler instance designate single class ASTs (one .java file can contain more than one class implementation)
- all classes ASTs are stored in List collection attribute

In next step the evaluation process begins. All calculations are handled by *calculate()* method and output data are stored in *Object[][]* attribute and then presented to the user by *addComponentToPane()* method.

The *calculate()* method takes a list of class' AST as a parameter. Based on information stored in ASTs method calculates metrics in following order:

In *for()* loop across all ASTs:

- creates CFS instance, which calculates Cognitive Functional Size
- creates HSM instance, which calculates metrics connected with Halstead's Software Science
- creates CC instance with methods ASTs from CFS object and calculate CC metric
- creates CK instance, which calculates Chidamber and Kemerer metrics
- all data are stored in *results* 2D array

The final step is data presentation. For this purpose I wrote *addComponentToPane()* method. The data are presented in two forms.

Basic one is a table. For data processing for table use *tableData()* and *tableHeaders()* methods are responsible. These methods returns data extracted from *results* array in form applicable for *JTable* constructor.

Chapter 6

Results interpretation and conclusions

In this chapter I will show how the SMT extension works. How the user can interpret the results and how to choose between examples.

6.1 Simple examples

In this section I will show an evaluation of three examples for date implementation.

The first two examples are from the "Beauty and the Beast" paper [13]. According to the report's authors the better example is Date_Beauty.java. As a third example I chose an date implementation from Java examples repository [12]. In the following figures are presented screen shots from SMT for every class (Figures 6.1, 6.2, 6.3) and on the last one the evaluation of the entire project (Figure 6.4).

SMT results : Class Date_Beauty

Measures | Charts

Measure:	Date_Beauty	Day	Month	Year	Average
Cognitive Functional Size	4	9	12	11	*9.0*
Average Cyclomatic Complexity	1.0	1.333	1.75	1.0	*1.27*
The biggest method's CC	1	2	3	1	*1.75*
number of unique operators	12	5	8	9	*8.5*
number of unique operands	8	1	3	1	*3.25*
total number of operators	47	9	17	11	*21.0*
total number of operands	96	16	32	14	*39.5*
size of the "vocabulary" used	20	6	11	10	*11.75*
length of the program\class	143	25	49	25	*60.5*
Volume	618.035	64.624	169.512	83.048	*233.804*
Level	0.02	0.05	0.058	0.039	*0.041*
Difficulty	48.0	20.0	17.066	25.2	*27.566*
Effort	29665.714	1292.481	2893.007	2092.814	*8986.003*
Weighted Methods per Class	2	3	4	4	*3.25*
Depth of Inheritance Tree	0	0	0	0	*0.0*
Response for a Class	0	2	2	1	*1.25*

Figure 6.1: Evaluation of the Date_Beauty.java class [13]

SMT results : Class Date_Beast

Measures | Charts

Measure:	Date_Beast	Average
Cognitive Functional Size	16	16.0
Average Cyclomatic Complexity	12.5	12.5
The biggest method's CC	24	24.0
number of unique operators	9	9.0
number of unique operands	4	4.0
total number of operators	50	50.0
total number of operands	72	72.0
size of the "vocabulary" used	13	13.0
length of the program\class	122	122.0
Volume	451.453	451.453
Level	0.018	0.018
Difficulty	54.0	54.0
Effort	24378.496	24378.496
Weighted Methods per Class	2	2.0
Depth of Inheritance Tree	0	0.0
Response for a Class	0	0.0

Figure 6.2: Evaluation of the Date_Beast.java class [13]

SMT results : Class DateExample

Measures | Charts

Measure:	DateExample	Average
Cognitive Functional Size	159	159.0
Average Cyclomatic Complexity	2.5	2.5
The biggest method's CC	7	7.0
number of unique operators	13	13.0
number of unique operands	18	18.0
total number of operators	115	115.0
total number of operands	142	142.0
size of the "vocabulary" used	31	31.0
length of the program\class	257	257.0
Volume	1273.228	1273.228
Level	0.048	0.048
Difficulty	20.511	20.511
Effort	26115.33	26115.33
Weighted Methods per Class	4	4.0
Depth of Inheritance Tree	0	0.0
Response for a Class	3	3.0

Figure 6.3: Evaluation of the DateExample.java class [12]

On the Figure 6.1 are results for Date_Beauty.java file. In this example are four classes. One Date_Beauty is a main public class. This class has a set of three subclasses: Day, Month and Year. Date_Beauty.java is an example of Object-Oriented decomposition. The Date_Beast.java and DateExample.java on the other hand enclose everything in one class.

The Cognitive Functional Size (CFS)

CFS for class Date_Beauty is just 4. However, we need to take under consideration all three subclasses. The Day has CFS equal to nine, Month twelve, and year eleven. In summary it will give us CFS equal to 36. However, we need to consider that all subclasses have similar construction. The main idea is similar. And the effort to understand all of them is actually equal the effort for understanding the first one plus the effort for adjusting parameters (variables names) in others. To this we must add CFS from the main class, which has a different construction.

In summary we can take the average value CFS from subclasses with similar construction.

The CFS for Date_Beast is 16. It is not much if we take a look on CFS for DateExample class(159). However, DateExample defines more functions than beauty and beast, and use Java Date class. But what I want to show is the difference in understanding simple date implementation with Date_Beauty and Date_Beast examples and more complicated DateExample class.

The Average Cyclomatic Complexity (CC) and The biggest method's CC

The Average Cyclomatic Complexity (CC) and The biggest method's CC shows that in Date_Beast are many decision blocks, especially in one method (with CC = 24). That implies that this example is hard to follow and suggests analyze many attributes at the time. The Date_Beauty.java gives CC on low level what suggest set of simple classes. The DateExample CC can be for many unexpected, however when you look on Weighted Method per Class you will see that in this example are 4 methods, where one has CC equal to 7. Results tell us that the example has not many linearly independent circuits and is decomposed on 4 methods. That is correct. DataExample has not many decisions blocks, but many function calls and that is why in this case CFS differs so much from CC.

The Halstead's science metrics

In case of example with subclasses (e.g. Date_Beauty 6.1) the main class contains results from all subclasses. In our example Date_Beauty class is the only one that needs to be taken under consideration during analysis. All metrics for the subclasses are already in the main class. In case of CFS or CC we have to count subclasses as a part of main class; in Halstead's metrics it is not necessary. Every subclass has their own metrics results and they can be analyzed as separate classes but they should not be treated as component for the main class' results. It is a result of AST construction (see Figure 5.1 and compare AST for Date_Beauty and AST for E (example class with inheritance in one file) class).

Lets analyze date classes:

The Date_Beast, as we can see (Figure 6.2), contains just nine unique operators and four operands (in default configuration, see User's Guide for details). Date_Beauty con-

tains 12 operators and 8 operands. As you can see there is no direct relation between operators/operands in subclasses. The result in main class is not a sum of operators/operands from subclasses and the main class. The reason is in AST construction (Figure 5.1). Operands or operators' names (identifiers) are equal in some cases, e.g. "day" in 3 classes is the same that is why it is counted as one operand (in case of **unique** operands).

DateExample (Figure 6.3) has vocabulary from 31 elements. It is eleven more then Date_Beauty. However, the difference is mainly in the number of unique operands (10). But, as I said before, this example is written to exercise most of the functionality of java.util.Date class. It contain many local variables to show differences between many Date class options. That is why the length of the class is so high; 257. There are many print and compare operations.

Lets put all information about "vocabulary" together and analyze them. The number of unique operators and operands summarize in size of the "vocabulary" used. Total number of operators and operands summarize in length of the class. These two metrics give us a valuable information about code variety. Code with smaller amount of variables is easer to understand. However, here comes the name convention in Java. If the program is written in Java names convention the amount of variables is not so important (variable's name determines meaning in the system).

SMT results : Project date

Measures | Charts

Measure:	Date_Beast	Date_Beauty	Day	Month	Year	DateExample	Average
Cognitive Functional Size	16	4	9	12	11	159	*35.166*
Average Cyclomatic Complexity	12.5	1.0	1.333	1.75	1.0	2.5	*3.347*
The biggest method's CC	24	1	2	3	1	7	*6.333*
number of unique operators	9	12	5	8	9	13	*9.333*
number of unique operands	4	8	1	3	1	18	*5.833*
total number of operators	50	47	9	17	11	115	*41.5*
total number of operands	72	96	16	32	14	142	*62.0*
size of the "vocabulary" used	13	20	6	11	10	31	*15.166*
length of the program\class	122	143	25	49	25	257	*103.5*
Volume	451.453	618.035	64.624	169.512	83.048	1273.228	*443.316*
Level	0.018	0.02	0.05	0.058	0.039	0.048	*0.038*
Difficulty	54.0	48.0	20.0	17.066	25.2	20.511	*30.796*
Effort	24378.496	29665.714	1292.481	2893.007	2092.814	26115.33	*14406.307*
Weighted Methods per Class	2	2	3	4	4	4	*3.166*
Depth of Inheritance Tree	0	0	0	0	0	0	*0.0*
Response for a Class	0	0	2	2	1	3	*1.333*

Figure 6.4: Evaluation of the Date project

The Volume, Level and Difficulty are results based on calculations on vocabulary data. What is interested that Data_Beauty has lower Difficulty than Data_Beast and DataExample has even lower Difficulty than Data_Beauty. This results say that the most Difficulty is Date_Beast, however Effort metric point that Date_Beauty need more effort that Date_Beast.

CK metrics

CK metrics give very useful data according to design complexity. The most important is WMC which says how many methods contain a class, and RFC which describe cor-

relation between methods within a class. The DIT can in Java take just 0 or 1 value, because in Java just one level of inheritance is allowed.

6.2 Restrictions

Implementation of SMT extension contain some formulas which require the Java version 1.6 to be installed on user's machine.

6.3 Limitations

The SMT will evaluate source code written in Java programming language version 1.3 or earlier. This tool does not have any mechanism for syntax error handling that is why it should be used on previously compiled projects.

6.4 Future work

When the ANTLR sympathizers provide an open source grammar for Java version 1.6 I will apply changes to SMT, that it will be able to evaluate source code written with newest version of Java.

When I will meet need from user for other metrics I will do my best to satisfy them all.

Chapter 7

Acknowledgments

I would like to thank my supervisor, Jürgen Börstler, for his good advice, his time, his help at each stage and for golden patience.

Thanks to all my friends and family for all your support and faith when the time was right.

I would like to dedicate my Master's Thesis Paper and my Master's Degree to my parents, Zofia and Marian Polok, as a gift for their 30th wedding anniversary, 17th of June 2008.

References

[1] John Lilley, John Mitchell, Terence Parr, Scott Stanchfield . Public Domain Java 1.3 grammar. http://www.antlr2.org/grammar/java.

[2] Esteves R J. Abreu Brito F., Goualo M. Towards the design quality evaluation of object-oriented software system. Proceeding of the 5th International Conference on Software Quality, Austin, Texas, USA.

[3] Apache. http:/hc.apache.org/httpclient-3.x/. 03.2008.

[4] Ashley J.S Mills. Ashley J.S Mills' very thorough tutorial at The University of Birmingham (Java output). http://supportweb.cs.bham.ac.uk/docs/tutorials/docsystem/build/tutorials/antlr/antlr.html.

[5] I. Bluemke. Metrics for assessing complexity and testability of object-oriented software. *Prodialog*, 13:1–13, 2001.

[6] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.

[7] Eclipse contributors and others. Eclipse official site. http://www.eclipse.org.

[8] Norman E. Fenton. *Software Metrics, A Rigorous Approach.* Chapman & Hall, London, England, 1991.

[9] Maurice H. Halstead. *Elements of Software Science (Operating and programming systems series).* Elsevier Science Inc., New York, NY, USA, 1977.

[10] Piotr Zajac Ilona Bluemke. Metryki mood w rational rose. Department of Computer Science, Politechnika Warszawska, Warsaw, Poland.

[11] Imagix Team. Imagix 4D official site. http://www.imagix.com.

[12] Jeffrey M. Hunter. Java Examples Repository. http://www.idevelopment.info/data/Programming/java/PROGRAMMING_Java_Programming.shtml.

[13] Jürgen Börstler, Micheal E. Caspersen, Marie Nordström. Beauty and the Beast,Toward a Measurement Framework for Example Program Quality. UMEÅ UNIVERSITY, Department of Computing Science, SE-901 87 UMEÅ SWEDEN.

[14] Coral Calero Marcela Genero, Mario Piattini. A survey of metrics for uml class diagrams. *Journal of Object Technology*, 4(9), November-December 2005.

[15] McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2:308–320, 1976.

[16] Michael Kölling. Writing Extensions For BlueJ. http://bluej.org/doc/writingextensions.html.

[17] D M Miller, R S Maness, J W Howatt, and W H Shaw. A software science counting strategy for the full ada language. *SIGPLAN Not.*, 22(5):32–41, 1987.

[18] Milosz Kmieciak, Kamil Krysztofiak. Zaawansowane projektowanie obiektowe. Metryki obiektowe. 2008.

[19] Object Management Group. Unified Modeling Language. http://www.uml.org.

[20] Jingqiu Shao and Yingxu Wang. A new measure of software complexity based on cognitive weights. *CAN. J. ELECT. COMPUT ENG.*, 28, April 2003.

[21] Sun Microsystems. Code Conventions for the Java^{TM} Programming Language. http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html.

[22] The ANTLR team. ANTLR ver 2 official website. http://www.antlr2.org.

[23] The BlueJ Team. BlueJ Extensions API. http://bluej.org/doc/extensionsAPI/.

[24] The BlueJ team. BlueJ official site. http://www.bluej.org.

[25] Linda Westfall. 12 steps to useful software metrics. The Westfall Team, 2005.

[26] Wikipedia, The Free Encyclopedia. Wikipedia - AST definition. http://en.wikipedia.org/wiki/Abstract_syntax_tree.

Appendix A

Source Code

Complete source code can be found in:
`http://www.cs.umu.se/~ens06jpk/thesis/SourceCode/`

Appendix B

User's Guide

B.1 Installation

SMT work as any other BlueJ's extension and installation process is identical. To get it to work just copy the smt.jar file from project repository (`http://www.cs.umu.se/~ens06jpk/thesis/`) to %BlueJ%/lib/extensions directory and restart BlueJ. The SMT extension should be now present in Help->Installed Extensions dialog in BlueJ (See Figure B.1).

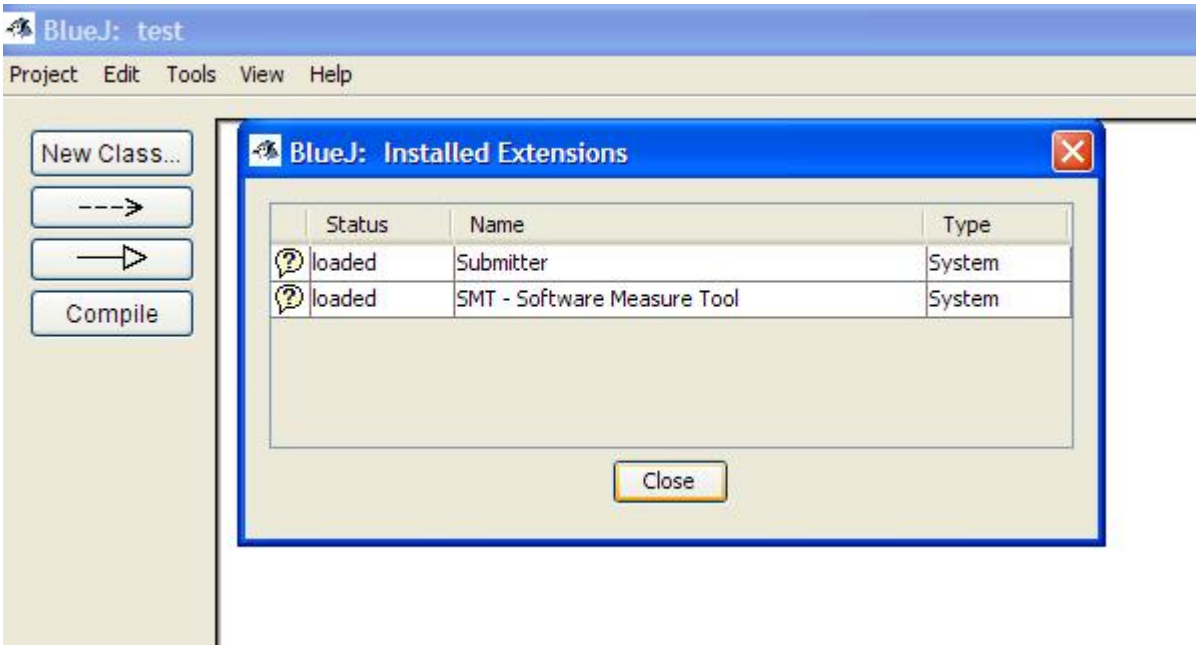


Figure B.1: Correct installed SMT is visible in BlueJ's Installed Extensions dialog

B.2 SMT's settings

SMT has a special dialog where the user can play with a set of parameters, which configure results presentation and counting rules. To run this dialog go into the BlueJ menu and choose "Tools" and then "SMT options"(See Figure B.2).

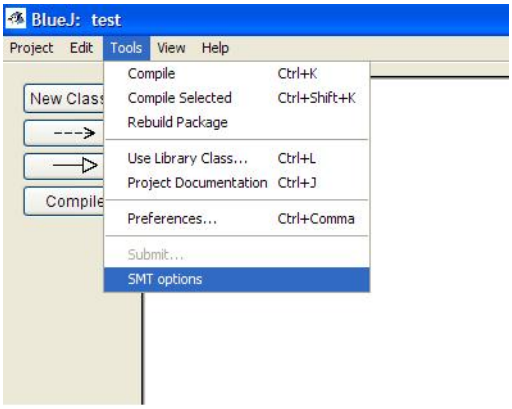


Figure B.2: SMT options access

The "SMT Options" window will appear (Figure B.3). The options window has two main tabs: "Metrics" and "Halstead operators settings".

The first tab contain all metrics. Every checked metric will be visible in "Results Window"(Figure B.7). Second tab contains parameters used in Halstead metrics calculations. Parameters are sorted in four groups:

- Basic - contain basic operators
- Arithmetic - contain arithmetic operators
- Java language - contain unique Java operators
- Others - contain other operators, e.g. brackets

Every "checked" operator is taken under consideration in counting rules of Halstead's metrics.

The SMT gives possibility to use others then predefined operators. This option is available in tab "Add new" (Figure B.4).

In field "Operator name" type operator name. This name will describe operator in "SMT options" window. In field "AST's node identifier:" type an integer value, which represents AST node ID. The list of all AST nodes ID can be found in "Java Token Types" chapter. Last options is to choose an operator group. Operator needs to belong to any of four groups of operators. To add a new operator press the "add" button.

B.3 Evaluation

SMT gives two ways of evaluation:

- Class evaluation

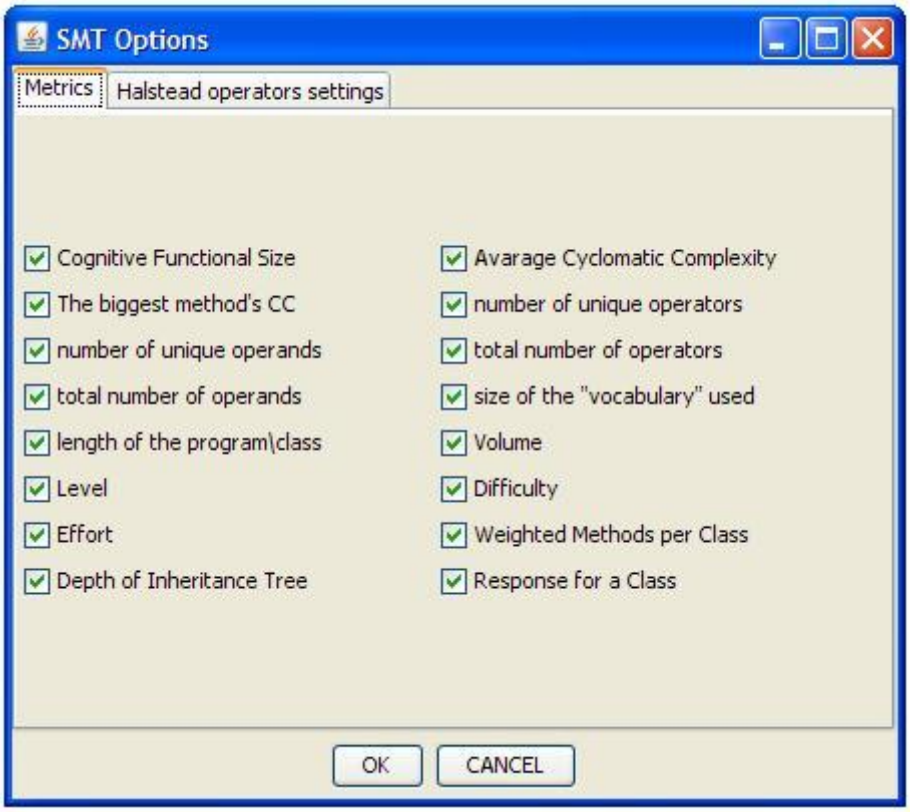


Figure B.3: SMT options window

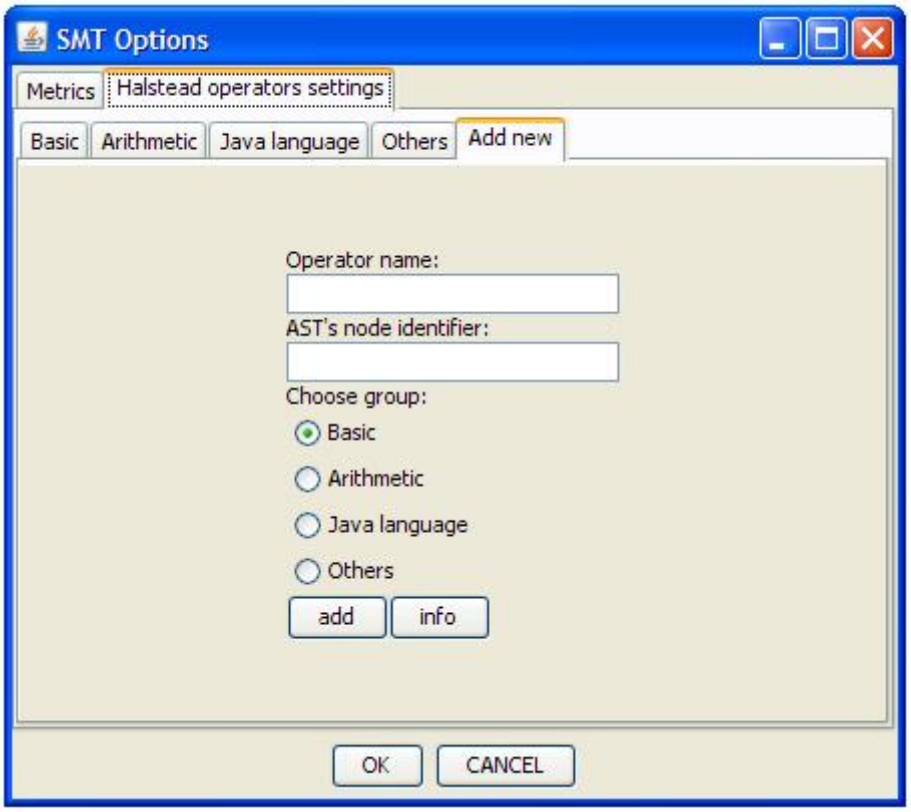


Figure B.4: SMT options window

– Project evaluation

First option evaluates only selected class, the seconds evaluate all classes in the project directory.

To call evaluation right click on target class and choose SMT->Evaluate Class/Evaluate Project (See Figure B.5).

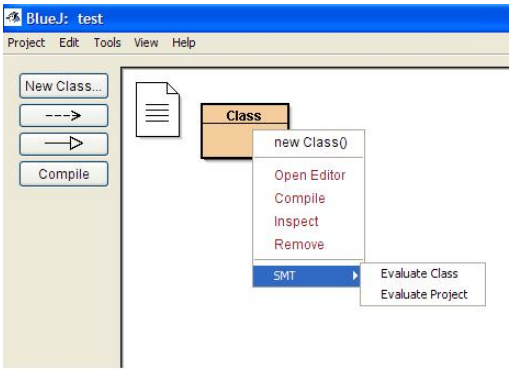


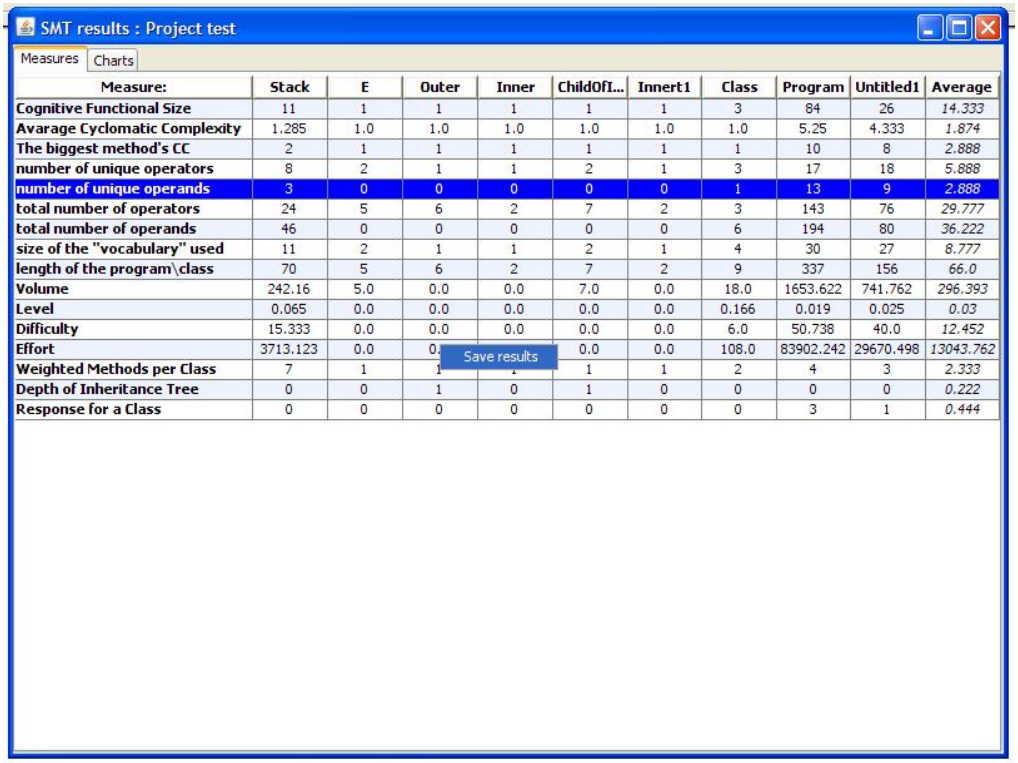
Figure B.5: Call evaluation

This will cause the SMT results window to appear (See Figure B.7).

All results can be saved into a file. To do so, right click on the result's table (See Figure B.6) and choose the save option. The standard Save File Dialog will appear. Choose file name and destination and click "Save" button.

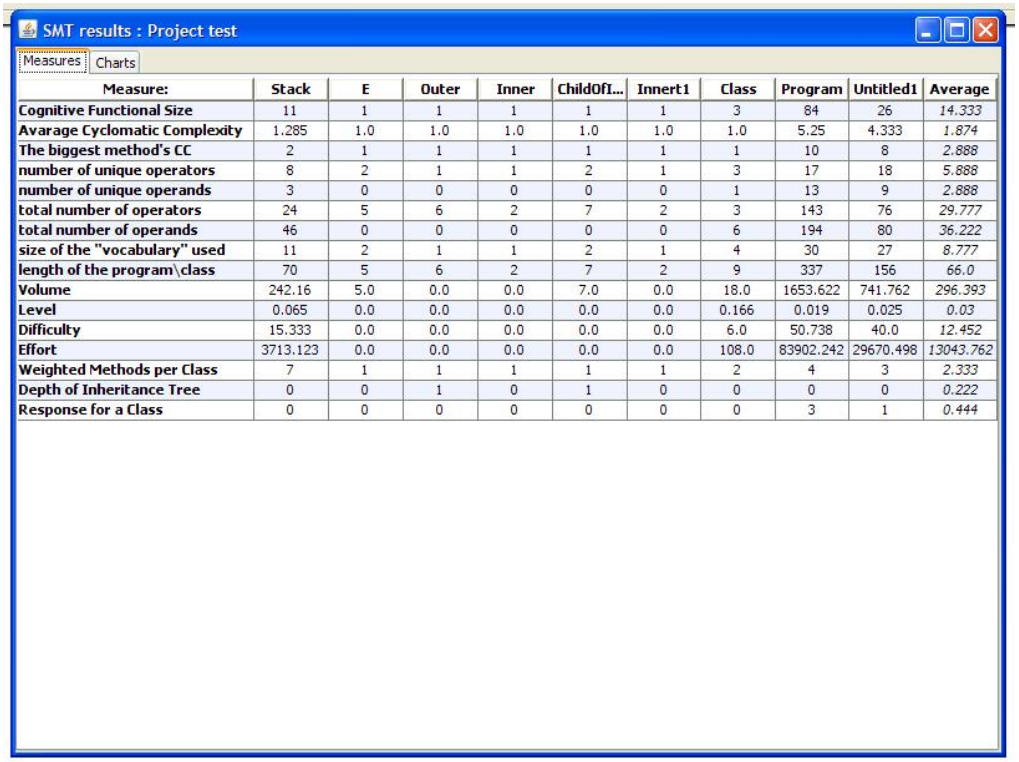
Data will be stored in text mode. First text line is an list of column names separated with semicolon. where the first one is always "Measure:" and the last one "Average". The in between ones are names of evaluated classes.

Every next line contains metric name and values separated with semicolon.



Measure:	Stack	E	Outer	Inner	Child0f1...	Innert1	Class	Program	Untitled1	Average
Cognitive Functional Size	11	1	1	1	1	1	3	84	26	*14.333*
Avarage Cyclomatic Complexity	1.285	1.0	1.0	1.0	1.0	1.0	1.0	5.25	4.333	*1.874*
The biggest method's CC	2	1	1	1	1	1	1	10	8	*2.888*
number of unique operators	8	2	1	1	2	1	3	17	18	*5.888*
number of unique operands	3	0	0	0	0	0	1	13	9	*2.888*
total number of operators	24	5	6	2	7	2	3	143	76	*29.777*
total number of operands	46	0	0	0	0	0	6	194	80	*36.222*
size of the "vocabulary" used	11	2	1	1	2	1	4	30	27	*8.777*
length of the program\class	70	5	6	2	7	2	9	337	156	*66.0*
Volume	242.16	5.0	0.0	0.0	7.0	0.0	18.0	1653.622	741.762	*296.393*
Level	0.065	0.0	0.0	0.0	0.0	0.0	0.166	0.019	0.025	*0.03*
Difficulty	15.333	0.0	0.0	0.0	0.0	0.0	6.0	50.738	40.0	*12.452*
Effort	3713.123	0.0	0.		0.0	0.0	108.0	83902.242	29670.498	*13043.762*
Weighted Methods per Class	7	1	1	1	1	1	2	4	3	*2.333*
Depth of Inheritance Tree	0	0	1	0	1	0	0	0	0	*0.222*
Response for a Class	0	0	0	0	0	0	0	3	1	*0.444*

Figure B.6: SMT save results



Measure:	Stack	E	Outer	Inner	ChildOfI...	Innert1	Class	Program	Untitled1	Average
Cognitive Functional Size	11	1	1	1	1	1	3	84	26	*14.333*
Avarage Cyclomatic Complexity	1.285	1.0	1.0	1.0	1.0	1.0	1.0	5.25	4.333	*1.874*
The biggest method's CC	2	1	1	1	1	1	1	10	8	*2.888*
number of unique operators	8	2	1	1	2	1	3	17	18	*5.888*
number of unique operands	3	0	0	0	0	0	1	13	9	*2.888*
total number of operators	24	5	6	2	7	2	3	143	76	*29.777*
total number of operands	46	0	0	0	0	0	6	194	80	*36.222*
size of the "vocabulary" used	11	2	1	1	2	1	4	30	27	*8.777*
length of the program\class	70	5	6	2	7	2	9	337	156	*66.0*
Volume	242.16	5.0	0.0	0.0	7.0	0.0	18.0	1653.622	741.762	*296.393*
Level	0.065	0.0	0.0	0.0	0.0	0.0	0.166	0.019	0.025	*0.03*
Difficulty	15.333	0.0	0.0	0.0	0.0	0.0	6.0	50.738	40.0	*12.452*
Effort	3713.123	0.0	0.0	0.0	0.0	0.0	108.0	83902.242	29670.498	*13043.762*
Weighted Methods per Class	7	1	1	1	1	1	2	4	3	*2.333*
Depth of Inheritance Tree	0	0	1	0	1	0	0	0	0	*0.222*
Response for a Class	0	0	0	0	0	0	0	3	1	*0.444*

Figure B.7: SMT results

Appendix C

Java Token Types

File contain list of all possible types of nodes in AST. File is a result of compilation "java.g" file with ANTLR.

```
// $ANTLR 2.7.5 (20050128): java.g -> JavaTokenTypes.txt$
Java    // output token vocab name
BLOCK=4
MODIFIERS=5
OBJBLOCK=6
SLIST=7
CTOR_DEF=8
METHOD_DEF=9
VARIABLE_DEF=10
INSTANCE_INIT=11
STATIC_INIT=12
TYPE=13
CLASS_DEF=14
INTERFACE_DEF=15
PACKAGE_DEF=16
ARRAY_DECLARATOR=17
EXTENDS_CLAUSE=18
IMPLEMENTS_CLAUSE=19
PARAMETERS=20
PARAMETER_DEF=21
LABELED_STAT=22
TYPECAST=23
INDEX_OP=24
POST_INC=25
POST_DEC=26
METHOD_CALL=27
EXPR=28
ARRAY_INIT=29
IMPORT=30
UNARY_MINUS=31
UNARY_PLUS=32
CASE_GROUP=33
```

```
ELIST=34
FOR_INIT=35
FOR_CONDITION=36
FOR_ITERATOR=37
EMPTY_STAT=38
FINAL="final"=39
ABSTRACT="abstract"=40
STRICTFP="strictfp"=41
SUPER_CTOR_CALL=42
CTOR_CALL=43
LITERAL_package="package"=44
SEMI=45
LITERAL_import="import"=46
LBRACK=47
RBRACK=48
LITERAL_void="void"=49
LITERAL_boolean="boolean"=50
LITERAL_byte="byte"=51
LITERAL_char="char"=52
LITERAL_short="short"=53
LITERAL_int="int"=54
LITERAL_float="float"=55
LITERAL_long="long"=56
LITERAL_double="double"=57
IDENT=58
DOT=59
STAR=60
LITERAL_private="private"=61
LITERAL_public="public"=62
LITERAL_protected="protected"=63
LITERAL_static="static"=64
LITERAL_transient="transient"=65
LITERAL_native="native"=66
LITERAL_threadsafe="threadsafe"=67
LITERAL_synchronized="synchronized"=68
LITERAL_volatile="volatile"=69
LITERAL_class="class"=70
LITERAL_extends="extends"=71
LITERAL_interface="interface"=72
LCURLY=73
RCURLY=74
COMMA=75
LITERAL_implements="implements"=76
LPAREN=77
RPAREN=78
LITERAL_this="this"=79
LITERAL_super="super"=80
ASSIGN=81
LITERAL_throws="throws"=82
```

```
COLON=83
LITERAL_if="if"=84
LITERAL_else="else"=85
LITERAL_for="for"=86
LITERAL_while="while"=87
LITERAL_do="do"=88
LITERAL_break="break"=89
LITERAL_continue="continue"=90
LITERAL_return="return"=91
LITERAL_switch="switch"=92
LITERAL_throw="throw"=93
LITERAL_case="case"=94
LITERAL_default="default"=95
LITERAL_try="try"=96
LITERAL_finally="finally"=97
LITERAL_catch="catch"=98
PLUS_ASSIGN=99
MINUS_ASSIGN=100
STAR_ASSIGN=101
DIV_ASSIGN=102
MOD_ASSIGN=103
SR_ASSIGN=104
BSR_ASSIGN=105
SL_ASSIGN=106
BAND_ASSIGN=107
BXOR_ASSIGN=108
BOR_ASSIGN=109
QUESTION=110
LOR=111
LAND=112
BOR=113
BXOR=114
BAND=115
NOT_EQUAL=116
EQUAL=117
LT=118
GT=119
LE=120
GE=121
LITERAL_instanceof="instanceof"=122
SL=123
SR=124
BSR=125
PLUS=126
MINUS=127
DIV=128
MOD=129
INC=130
DEC=131
```

```
BNOT=132
LNOT=133
LITERAL_true="true"=134
LITERAL_false="false"=135
LITERAL_null="null"=136
LITERAL_new="new"=137
NUM_INT=138
CHAR_LITERAL=139
STRING_LITERAL=140
NUM_FLOAT=141
NUM_LONG=142
NUM_DOUBLE=143
WS=144
SL_COMMENT=145
ML_COMMENT=146
ESC=147
HEX_DIGIT=148
EXPONENT=149
FLOAT_SUFFIX=150
```