Contents

Chapter 1

Introduction

Chapter 2

Basic terms

This chapter introduces and explains the basic terms that have been used in the thesis.

Software metrics

Software metrics are special kind of measurements specified for software used to identify quality of code, the complexity of the system and cost of maintenance and flexibility. The purpose of software metrics is to investigate and evaluate the quality of provided software and encourage developers to improve the quality of provided products [5].

Software engineering

Software engineering provides the theoretical foundations for building software and focuses on implementing the software in a controlled and scientific way. It describes the collection of techniques that apply an engineering approach to the implement and support of software products. Software engineering activities include managing, costing, planning, modelling, analysing, specifying, designing, implementing, testing, and maintaining. Engineering approach means that each activity is understood and controlled,

so that there could be no surprises as the software is specified, designed, built, and maintained [4].

Object-oriented metrics

The rise in popularity of object oriented programming create a need to prepare metrics that are able to measure and evaluate aspects characteristic for objectivity: inheritance, polymorphism, encapsulation, cohesion and coupling. First set of object-oriented metrics were proposed by S.R. Chidamber and C.F. Kemerer [1].

Object-oriented programming paradigm

Object-oriented programming paradigm is a concept of objects that have attributes (fields) and associated behaviour (procedures, methods, functions). Generally, the object is a instance of class. Class is a representation of abstraction and its instances are used to interact with each other. Nowadays, the most popular object-oriented programming languages are C++, Java, C Sharp and Objective-C.

> *Object-oriented programming is a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships* [3].

Object-oriented programming makes code easier to organize, understand and managed. The modification of code could be applied without affecting other aspect of system. It is easier to introduce update and upgrades and when the system grows over the years it is more manageable. It allows also to split the responsibility of functionality of code by several developers, so they could work individually for one system but for another components of this system [3].

Inheritance

Inheritance is a relation between classes where one class (called child, derived or subclass) is extended by another class (called parent, base or super class). This relationship rises a hierarchy where subclasses could inherit attributes and behaviours (methods) from pre-existing classes. The semantics of class inheritance could vary from language to language, but commonly the subclass automatically inherits the instance variables and member functions of its superclass. Some languages enable to inherit from many classes and the access to attributes and methods from parent class are defined by set of access modifiers (for example in Java language are public, private, protected and default modifiers). The superclass defines a common interface and foundational functionality, which specialized subclasses could inherit, modify, and implement. The methods and attributes inherited by a subclass could be reused in the subclass. A reference to an instance of a class might be referring one of its subclasses. The actual class of the object being referenced is impossible to predict at compile-time. A uniform interface is used to invoke the member functions of objects of a number of different classes. Subclass might replace superclass functions with entirely new functions by naming its method the same as methods in super class (it is called overriding). In some languages a class could be defined as unable to be inherited by adding to class special declaration (for Java it is a *"final"* keyword and *"sealed"* for C Sharp) Such word added to declaration of class restricts the re-usability of all methods and attributes of this class. Similar functionality could be also applied to some attributes or methods from given class. Introducing inheritance mechanism enabled to limit number of duplicated code and separate behaviours from super class and enable to detail it in subclass [7].

Polymorphism

Polymorphism refer to the biological principle in which organism or species could have different forms or stages. This principle has also its use in object oriented programming. Subclass of parent class can define their own unique behaviours (methods) and what is more, share the same of functionality of the parent class [12].

There are different kinds of polymorphism:

- *"ad hoc" polymorphism* is supported by many languages as function overloading and it means that method could be applied to arguments of different types.
- *parametric polymorphism* is marked when code is written without mentioned of any specific type and thus could be use with any number of types.
- *inclusion polymorphism (sub typing)* is a concept where a name might denote instances of many different classes as long as they are related by some common superclass [3].

General, polymorphism could refer to *"many shapes"*. It is means ability to request the same operations, but performed by a wide range of different types of things. In OOP, the polymorphisms is achieved by using many different techniques named method overloading, operator overloading and method overriding.

Encapsulation

Encapsulation is a mechanism for restricting access to some of object's methods and attributes. It *"is the process of compartmentalizing the elements of an abstraction that constitute its structure and behaviour; encapsulation serves to separate the contractual interface of an abstraction and its implementation"* [3]. This mechanism provides explicit kind of barrier which leads to a clear separation of access and non-accessible part of implementation. In practise it means that class should have two parts: an interface and implementation. The interface of the class presents only scheme, layout or list of instance of class behaviour. The implementation asserts the mechanism that allows realization of class behaviour.

Another important assumption of encapsulation is hiding class arguments. It is realised by forbidding direct access to class's arguments (using access modifiers) and providing

methods allowing setting and getting its values. It prevents unauthorized parties direct access to them [7].

Coupling

Coupling is the measure of the strength of association established by a connection from one class to another class. Strong coupling complicates a system because system is harder to understand, revise, change, or correct. This complexity could be reduced by designing system with weak dependency between classes.

Coupling is the term borrowed from structural programming but is also applicable to object-oriented analysis and design. However the concepts of coupling does not refer to inheritance, (inheritance is in certain sense a type of coupling) because introduces significant improvement of presenting classification abstractions and dependencies in object oriented paradigm [3].

Cohesion

The idea of cohesion is also taken from structural programming. Cohesion measures the degree of connection between objects in a single component. The one form of cohesion is coincidental cohesion where entirely unrelated abstractions are thrown into the same class, package or component. Another form of cohesion is functional cohesion. Elements of a class or component work together to provide some well-bounded behaviour [3].

Artefacts

Artefacts are used as the simplest form of measurement of source code. As the artefact it could be treated line(s) of code, field(s) of code, methods, classes, interfaces, components, packages. Basing on such artefacts, the quality of system could be evaluated.

Graph theory

Simple graph

A simple graph G is defined as a pair of sets (V, E), where:

- V is a finite non empty set of vertices, and
- E is a set of pairs of vertices, called edges.

It is often represented by $G = (V, E)$ [2].

Chain

In a simple graph $G = (V, E)$, a chain c is a finite sequence of vertices: $v_0, v_1, \cdots, v_m$ such that, for all i with $0<=i<m, v_i, v_{i+1}$ is an element of E.

- chain is represented by $c = [v_0, v_1, \cdots, x_m]$.
- length of the path c is the integer m, and it is represented by $l = m(c)$ ([1]).

Connected graph

A graph G is connected if, for all x and for all y [vertices], there exists a chain connecting x and y. A graph that is not connected could be divided into connected components.

- cycle graph - path that begins and ends with the same vertex.
- simple cycle - cycle that has a length of at least 3, and in which the beginning vertex only appears once more, as the ending vertex, and the other vertices appear only once ([1], [2]).

Directed graph

A directed graph (or digraph) is a set of vertices and a collection of directed edges that each connects an ordered pair of vertices. It is said that a directed edge points from the first vertex in the pair and points to the second vertex in the pair. It is used the names 0 through V-1 for the vertices in a V-vertex graph [9].

Strongly connected graph

Strongly Connected Graph is kind of a directed graph that has a path from each vertex to every other vertex [1].

Integrated Development Environment (IDE)

Integrated Development Environment (IDE) is a software application that is used to create, modify, and test the software. They provide complex functionality that enable to edit source code, compile or interpreter the code, provide debug tool, create graphical user interface, create database and components. One of most desirable feature of IDE is integration with intelli-sense mechanism.

The concept of IDE evolved from simple command based software which was not as useful as menu-driven software. Modern IDEs are mostly used in the context of visual programming, where applications are quickly created by moving programming building blocks or code nodes that generate flowchart and structure diagrams, which are compiled or interpreted.

Selecting a good IDE is based on factors, such as language support, operating system (OS) needs and costs associated with using the IDE etc.

There are several popular IDEs:

- Microsoft Visual Studio – definitive IDE for Windows application development covers languages like C++, C Sharp and VB.NET
- Netbeans – open source software providing programming tools for Java language
- Eclipse – firstly it was framework written in Java used to create rich client application, later rebuild to create application in Java. Various set of plug-ins extends its functionality for other languages.

- IntelliJ – it is a Java IDE by JetBrains, available as an Apache 2 Licensed community edition and commercial edition.

Chapter 3

Theoretical approach to software metrics

This chapter describes the set of commonly use software metrics. It starts from the description of fundamentals of metrics which are measurements. As a next step to understand the metrics, the Software Quality Model has been introduced. It explains the origin of the metrics. The described metrics have been also divided into groups.

3.1 Measurements

> **Measurement** is the process by which numbers or symbols are assigned to attributes of entities in the real world in such way as to describe them according to clearly defined rules [4].

Thus, measurement captures information about *attributes* and *entities*. An *entity* refers to the objects (such as person, things or Java class) or an event (software testing) in the real world. The entities are described by identifying characteristics and important attributes that differs one entity from another. An *attribute* is a feature or property

(method or attribute of class) of the entity. For example, the typical attributes could be a color of thing, cost of trip, elapsed time of testing software.

Describing the entities by attributes is often used by symbols or numbers. For instance, price is designated as a number of euros, while time of program execution in terms of seconds. Similarly, the satisfaction with national football team may be "very good", "good", "medium", "low", "very low". These numbers and phrases reflect human perception of real world. Thus, having to compare the prizes of iPhone and Samsung Galaxy, it is possible to state that one is more expensive than second one, or the attributes (features) of both differs.

Another term that refers measurement is *calculation*. Calculation is form of interpreting measurement and combing them into a quantified item that reflects some attributes whose value is tried to be understand. For example, the trainer could choose the players for next match basing on result of calculations. These calculations are made basing on attributes of every players and its possibility for facing the opponents attributes. Event though, the results of these calculations do not guarantee winning the champions, but , fortunately, in case of software engineering the results are rather clear.

Nowadays, the measurements become an significant part of software project management. Developers and customers prefer rely on measurement-based charts and graphs to support them what the visibility of project is and if the project is on right track. In order to compare and contrast projects, many companies and organization had created and defined set of standards measurements. It has been done so, because average client is not familiar with software terminology, so measurements could present a picture of progress in general and using understandable terms. For example, when a automotive company asks a software developer to write Computer Aided Design (CAD) document version control system, the customer usually knows a lot about CAD documents, but has no knowledge about compilers, programming languages and hardware needed to run designed software. The measurement should be presented in a way that explain both customer and developer what is progress of project and what the visibility for next days or months is.

3.2 Software metrics division

The basic division of software metrics relates to the method of analysis and testing. The metric could be divided into static and dynamic metrics. Static metrics analyse source code while dynamic determines, separately from the code and testing, the behaviour of a running program. A separate category of metrics are not directly related to software implementations, but the specification requirements of the customer as well as the course of the tests.

The static metrics could be divided into sub groups: size metrics, complexity metrics, object-oriented metrics and package metrics. They allow for testing the quality of source code by software developers. These metrics depicts the piece of code needed to be tested or simplify.

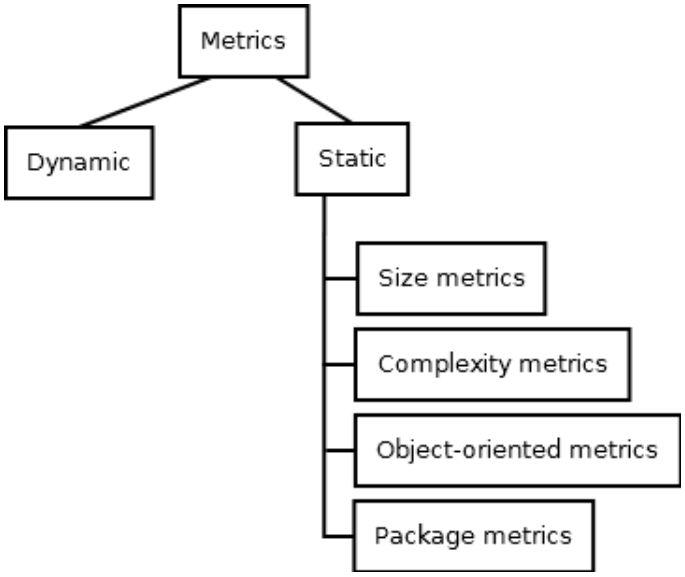

Figure 3.1: Static and dynamic metrics

The metrics could be also divided into three categories: product metrics, process metrics and project metrics. Product metrics describe the characteristics of the product such as size, complexity, design features, performance, and quality level. Process metrics are used to improve maintenance and development of software, for example: the effectiveness of defecting removal during development, the pattern of testing defect arrival, and the

response time of the fix process. Project metrics describe the project characteristics and execution, for example: the number of software developers, the life cycle of the software, cost, productivity, and schedule. Metrics could belong to multiple categories. For example, the quality metrics are both process metrics and project metrics [5].

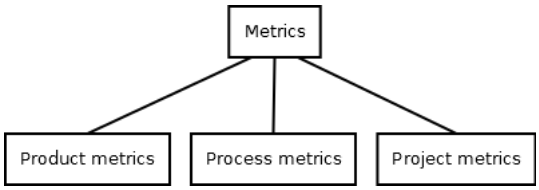

Figure 3.2: Metrics division

3.3 Software Quality Model

Experienced software developers are able to create a high quality software, however it is possible only when from the beginning all requirements and expectations are clearly defined. Having the overall view it is possible to design the system that has understandable and minimal complex structure.

However, the situation often changes after first release. It could happen so, because the client expectations were different or have changed, or new functionality need to be added, or the requirements have been clarified. In that case, the system loses its quality and get out of control. Testing, maintaining and extending software become a nightmare for developing team.

In order to keep development projects on the right track, and decrease the consequences of situations described above, several general software quality models gained the acceptance within the software engineering community. First people that described quality models are McCall and Boehm [4]. Figure 3.3 presents Boehm‘s view, while Figure 3.4 illustrates how McCall viewed quality.

These models focus on the final product (the executable code) and identify the attributes of quality from users point of view. The attributes are called *quality factors*. Because the

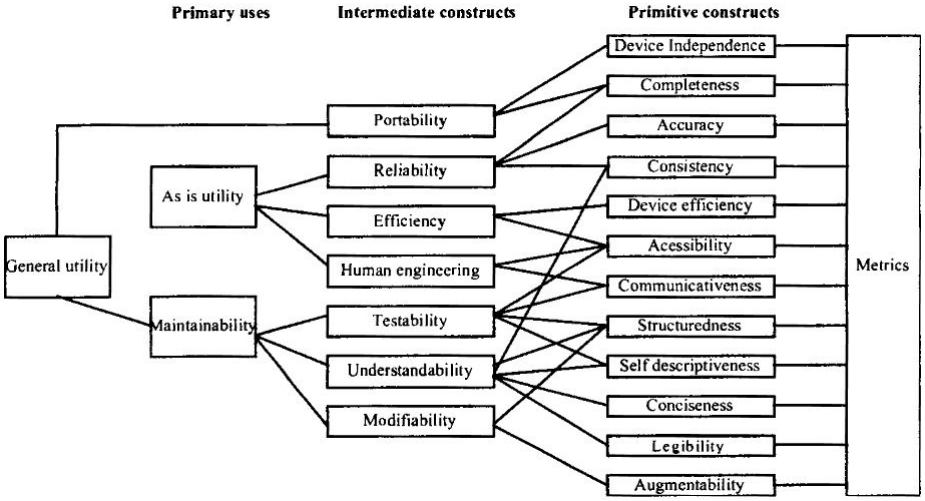

Figure 3.3: Boehm software quality model (image source: [4])

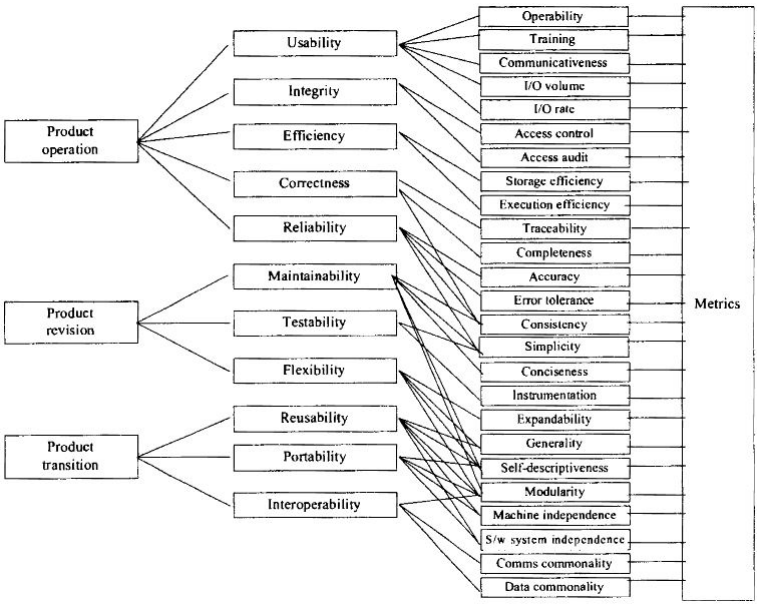

Figure 3.4: McCall software quality model (image source: [4])

quality factors names are still too general and no meaningful, they are decomposed into lower-level attributes called *quality criteria*. A further level of decomposition is required when quality criteria are connected to set of low-level, directly measurable attributes called *metrics*.

From derivation of McCall model, the basic international standard for software quality

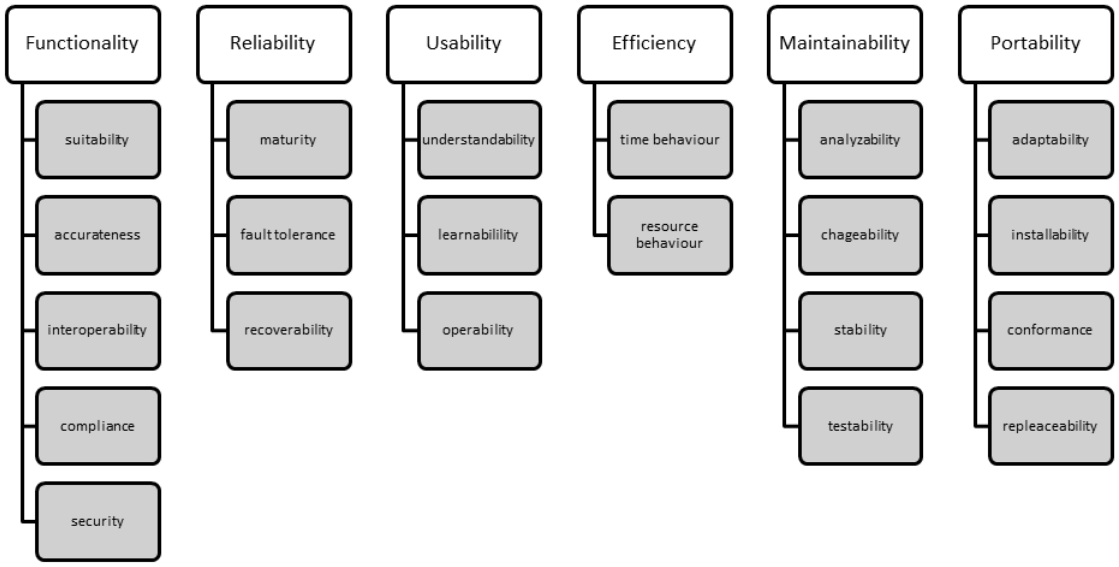

Figure 3.5: The ISO 9126 standard quality model

measurement was defined. The standard is more commonly referenced by its assigned standard number, ISO-9126. It defines software quality as *"the totality of features and characteristics of a software product that bear on its ability to satisfy stated or implied needs"* [11]. The six factors decompose the quality into:

- functionality
- reliability
- usability
- efficiency
- maintainability
- portability

3.3.1 Functionality

Functionality is the main purpose of any software. For certain type of software is relatively easy to define its functionality. However, the more functions a software has, the more complicated it becomes. For example, sales order processing systems should be able to record customer information so that it can be used to reference a sales order.

The main point to note is that functionality is expressed as a totality of essential functions that the software product provides. The presence or absence of some functions in a software product can be verified as either existing or not (given function could be or not be found in software product). The other software lcharacteristics present only some degree, because it cannot be simple stated that it characteristic is presented or not. The ISO-9126 quality model does not help measure overall process but only the software component. The relationship between software functionality within an overall business process is outside the scope.

There are five software quality criteria that characterize the usefulness of the software in a given environment (Table 3.1):

quality criteria	**explanation**
suitability	appropriateness to specification, functions of the software.
accurateness	correctness of the functions
interoperability	ability of software components to interact with other components
compliance	compliant capability of software
security	unauthorized access to the software functions

Table 3.1: *Functionality quality factor*

3.3.2 Reliability

Reliability is defined as capability of the system to being maintained and its service provision under defined conditions for defined periods of time. From one side of this

characteristic is fault tolerance that is the ability of a system to withstand component failure. For example if the connection to the server cannot be established for 30 seconds then system informs the users and follows failure procedures (Table 3.2).

quality criteria	**explanation**
maturity	frequency of failure of the software
fault tolerance	ability of software to withstand (and recover) from failure
recoverability	ability to bring back a failed system to full functionality

Table 3.2: *Reliability quality factor*

3.3.3 Usability

Usability is defined with regard to functionality and refers to the ease of use for a given function. For example, the most common functions are exposed on the ribbon to be easily access. The ability to learn how to use a system is also a major sub characteristic of usability. The Table 3.3 depicts the quality criteria of usability:

quality criteria	**explanation**
understandability	easiness of system understanding relating to human-computer interaction methods
learnability	learning effort for target users
operability	ability of using software in given environment

Table 3.3: *Usability quality factor*

3.3.4 Efficiency

Efficiency is concerned with the system resources used to provide required functionality. For example: amount of disk space, memory, network etc. provides a good indication of this characteristic (Table 3.4).

quality criteria	explanation
time behaviour	response time for user input
resource behavior	refers to hardware resources: cpu, disk, network usage

Table 3.4: *Efficiency quality factor*

3.3.5 Maintainability

Maintainability is characterized as ability to identify and fix a fault within a software component. It is impacted by code readability or complexity as well as modularization. Quality criteria that help with identification of the cause of a fault and then fixing the fault is the concern of maintainability. One of the sub characteristics of maintainability is also the ability to verify (or testing) a system (Table 3.5).

quality criteria	explanation
analyzability	ability to indentify cause of failure
changeability	amount of effort need to change a system
stability	impact of system after change
testability	effort needed to verify a system change

Table 3.5: *Maintainability quality factor*

3.3.6 Portability

Portability is characteristic referring to how well the software could adopt to changes in its environment or with its requirements (Table 3.6).

3.3.7 Software quality model summary

Basing on the measurement aspects of quality model, several separate definition could be stated. This definitions reflect the software usage or realities of system testing and, what is more, could be treated as software metrics.

quality criteria	explanation
adaptability	ability of change after adding extension
installability	effort needed to install software
conformance	ability to adjust after changing external component (i.e. database)
replaceability	ability to replace any component with other

Table 3.6: *Portability quality factor*

The simplest example is portability term expressed as:

$$Portability \quad = \quad 1 - \frac{ET}{ER} \tag{3.1}$$

where ET is a measure of the resources needed to move the system to the target environment and ER is a measure of the resource needed to create the system for a resident environment.

The interpretation of measurements based on quality factors described in ISO-9126, McCall or Boehm models is rather subjective. The objective measurements are more preferable, however the subjective opinion is better than no feedback.

The comprehensive picture of software quality is presented by six factors. Normally, the measurement should follow by some defined standards, however, in that case, the methods of measurement and interpretation are defined by end users and developers [4].

3.4 Size metrics

Each product of software development could be treated as physical entity and it could be described in terms of its size. It is straightforward and relatively simple to measure.

3.4.1 Lines of Code

Lines of Code (LoC) is the simplest metric used to measure the size of the program by counting the lines of code. It is the oldest and most widely used size metric. It is very simple concept that have its basics in Assembler programming where every physical line of code was presenting one instruction. Nowadays, in high level programming languages, the concept have broken down, because of differences between physical lines and instructions statements. As a result, the several variations have been created in order to improve basic LoC:

- LoC that counts only executable lines,
- LoC that counts executable lines plus data definitions,
- LoC that counts executable lines, data definitions and comments,
- LoC that counts executable lines, data definitions, comments, and job control language,
- LoC that counts lines as physical lines on a input screen,
- LoC that counts lines as terminated by logical delimiters [5].

There are two major types of LoC measurement: physical Source Lines of Code (SLoC) and Logical Lines of Code (LLoC). The specific definition of them are varied. SLoC is a line counter in the text of the program's source code including comment and blank lines. LLoC attempts to measure number of executable statements, but specific for given computer language. SLoC is sensitive to irrelevant code formatting and depends of the programmer code style convention.

LLoC is better metrics, because it estimates the complexity of a single file, class or procedure. Since LLoC is not affected by comments, blanks or line continuation, it is a supportive way to measure the amount of the actual programming work. A program

with a higher LLoC does more than a program with a lower LLoC. By adding features, the LLoC increases. By deleting features, the LLoC should decrease.

The main advantages of LoC is automatic counting, however used only for a specific language, because it cannot be used for other languages due to the syntax and structural differences among languages. LoC metric is also very intuitive because the effect of it could be visualized. However, there are many disadvantages that underlines the simplicity of this metrics. Using LoC it is not possible to measure the productivity of a project. The final number of lines in program is dependent to the experience of the developer, for instance, the same functionality could be rewritten by experienced developers using smaller number of lines of code. What is more, any form of LoC does not take into consideration code redundancy. There is no standard definition of what a line of code is. It is not clearly defined whether data declarations are included or does the comments count or what happens if a statement extends over several lines.

In relation to LoC, several additional terms has been also defined:

- KLOC can be used to measure thousands of LoC.
- KDLOC: 1,000 delivered lines of code
- KSLOC: 1,000 source lines of code
- MLOC: 1,000,000 lines of code
- GLOC: 1,000,000,000 lines of code

Summing up, the lines of code metric is one of the oldest and most common forms of software measurement, however is ineffective at comparing the same piece of code of implemented functionality in different languages. It cannot measure the productivity of programmer and code quality [5].

3.4.2 Code coverage

Edsger Dijkstra said: *"Testing never proves the absence of faults, it only shows their presence"*. The code coverage analysis is a measurement used in software testing to describe the degree to which the source code of a program has been tested. There are number of criteria (metrics) that determines how well the tests exercise the code.

The main purpose of code coverage analysis is not only finding the areas of a program not exercised by set of tests, but also creating additional test to increase the coverage or just to identify redundant test cases that does not increase coverage. Using code coverage analysis, it is also important to establish the minimum and maximum percentage of coverage in order to determine when to stop analysing the coverage.

Code coverage analysis is kind of structural testing technique (other names: glass box testing and white box testing). Structural testing compares behaviour of test program against the direct intention of the source code. This is a contrast to functional testing (other name: black-box testing) that compares behaviour of test program against a requirements specification. Structural testing examines how the program works, taking into account possible faults in the structure and logic. Functional testing examines what the program accomplishes, without regarding to how it works internally.

There are variety of coverage metrics. The paragraphs below contain a description of some fundamental metrics, their strengths, weaknesses and issues.

Statement coverage metrics reports every executable statement that is encountered. Note that a statement does not necessarily correspond to a line of code. Control-flow statements, such as `if, for` and `switch` are covered if the expression controlling the flow is covered as well as all the contained statements. Implicit statements, such as an omitted return, are not subject to statement coverage.

Advantage of statement coverage is verification of what the written code is expected to do and not to do. It also measures the quality of written code, checks the flow of different

paths in the program and it also ensures, if those path are tested or not.

Disadvantage of statement coverage is fact that it cannot test the false conditions. It does not report if the loop reaches its termination condition and it does not understand the logical operators.

Statement coverage is also called *statement execution, line, block, basic block* or *segment coverage.*

It could be quite difficult to achieve 100% statement coverage, because it might be sections of code designed to deal with error conditions, or rarely occurring events. There may also be code that should never be executed.

The statement coverage can be calculated as shown below:

$$Statement \quad coverage \quad = \quad \frac{number \quad of \quad statement \quad exercised}{total \quad number \quad of \quad statements} \cdot 100\% \qquad (3.2)$$

Decision coverage metrics also known as *branch coverage* or *all-edges coverage.* It covers both the true and false conditions. A branch is considered as the outcome of a decision and it simply measures which decision outcomes have been tested. This metrics takes a more in-depth view of the source code than simple statement coverage.

A decision is taken in if-statement, a loop control statement or a switch statement and the result of these statements could be either TRUE or FALSE.

The advantages of decision coverage is possibility to validate all the branches in the code, because all of them are reached. It is done in order to ensure that no branches lead to any not normal program‘s operation. This metric eliminates also problems that occur with statement coverage testing.

The decision coverage can be calculated as given below:

$$Decision \quad coverage \quad = \quad \frac{Number \quad of \quad decision \quad outcomes \quad exercised}{Total \quad number \quad of \quad decision \quad outcomes} \cdot 100\% \qquad (3.3)$$

The main goal of decision coverage is to ensure if a program could jump and it jumps to all possible destinations. The most simple example is a complete if statement:

```
1 if (x) {
2   print "a";
3 } else {
4   print "b";
5 }
```

Listing 3.1: Decision coverage metric passes

This coverage is only achieved here only if `x` is true on one occasion and false on another.

Condition coverage metric reports the true or false outcome of each condition separately. It measures the conditions independently of each other. It is similar to decision coverage but has better sensitivity to the control flow. During boolean expression evaluation it is need to ensure that all the terms in the expression are exercised. For example:

```
1 if (x || y) {
2   //...
3 }
```

Listing 3.2: Simple conditional coverage

In order to achieve the full condition coverage of this expression, the values of x and y are set to each of the four combinations of values they can take:

```
x=true; y=true;
x=true; y=false;
x=false; y=true;
x=false; y=false;
```

The condition coverage gets complicated, and difficult to achieve, as the expression gets complicated. For this reason there are a number of different ways of reporting condition coverage which try to ensure that the most important combinations are covered without worrying about less important combinations.

```
1 if (a || b) && c {
2   //...
3 }
```

Listing 3.3: Complex conditional coverage

The condition will be satisfied by the following set of tests:

```
a=true, b=true, c=true
a=false, b=false, c=false
```

However, it does not satisfy the condition coverage, because it need to test rest of possibilities:

```
a=false, b=false, c=true
a=true, b=false, c=true
a=false, b=true, c=true
a=false, b=true, c=false
```

Condition coverage is also known as *expression, condition-decision* and *multiple decision coverage.*

Path coverage metrics reports if every possible path in given function has been followed. This path is a unique sequence of branches from the function entry to the exit. It tests all possible combinations of logical conditions.

The purpose of path coverage metrics is to ensure that all paths through the program are taken. In sizeable program there will be an enormous number of paths through the program could be executed, so in practice the paths could be limited to those within a single function whether the function is not too big, or i.e. has simply two consecutive branches.

The main advantage of path coverage metrics is accuracy. However, from the other side,

the number of path increase exponentially to the number of branches and it is impossible to exercise all of time in small unit of time.

Other coverage metrics

There are also many other variation of coverage metrics like **function coverage metric** that reports when function or procedure is invoked. The **call coverage metric** reports every function call. The **loop coverage metric** informs whether each loop body has been executed zero times, exactly once or more than once. For supporting multi threading the **race coverage** reports whether multiple threads execute the same code at the same time. It supports with detecting failures with synchronization of resource access. The **relational operator coverage** reports if the boundary situations occur with operators: $<$, $<=$, $>$, $>=$,$==$.

Coverage metrics summary

The result of using statement coverage, decision coverage, or condition coverage should attain 80%-90% coverage or more before releasing. It does assure quality and the effort sacrificed to reach this goal, might be supportive in finding more bugs. The good project avoids setting a goal lower than 80%.

The coverage analysis help eliminate gaps in a test suite. It does not only provide accurate measurements in order to be useful, but also encourage software developers to analyse those results to the minimum detail and support them in decision how code could be improved ([10], [13]).

3.4.3 Code size metrics summary

Internal attribute of product like size is important and measuring it directly allows to predict the complexity. Number of characters, lines of code or code coverage is taken

into account as a determinant for these kind of metrics.

Basing on LoC it is possible to calculate derivative metrics like productivity quality or *"documentation–size"* metrics:

$$productivity = \frac{KLOC}{man - month} \tag{3.4}$$

$$quality = \frac{bugs}{KLOC} \tag{3.5}$$

$$documentation \quad size = \frac{documentation \quad pages}{KLOC} \tag{3.6}$$

It is worthy to mention that these kind of metrics are only supportive way of monitoring improvement of software development and it is not allowed to evaluate the code developers basing on results of these metrics.

3.5 Complexity metrics

Computational complexity has also impact on the software, so if the complex algorithms are used in software, there is a need to control the overall efficiency of program execution. The complexity metrics provide the information how complex the implemented methods in code are. The sections below describes the basic and commonly used complexity metrics.

3.5.1 McCabe's cyclomatic complexity

McCabe has based its metrics on analysis of some measurement concepts in graph theory. He has transferred these concepts into domain of software measurement. The terms

connected to graph theory have been explained in Chapter 2 in section: *Graph theory*. The term *cyclomatic number* is explained below (definition from: [1]).

> *The Cyclomatic Complexity (CC) number* ***v(G)*** *of a strongly connected directed graph is equal to the maximum number of linearly independent cycles.*

$$v(G) = e - v + p \tag{3.7}$$

where:

- G represents the graph,
- e represents the edges of the graph,
- v represents the vertices,
- p represents separate components.

The entity being measured is therefore a *"strongly connected directed graph"*. In software, the program is modelled with usage of *control flow graph* that represents abstract structure. In this abstract representation of a program or a procedure, each vertex in the graph represents a basic block, directed edges are used to depicts the jumps in the flow control. There are also two additional designed blocks: entry block through which control flow enters the flow graph and the exit block through which all control flow leaves.

The Figure 3.6 represents the exemplary flow control graph.

In the case of software, the control flow graph is transformed into a strongly connected graph, so graph cyclomatic number could be formulated as:

$$v(G) = e - v + p + (1 \quad virtual \quad edge) \tag{3.8}$$

Furthermore, the individual modules are also taken into consideration, instead of the whole software entity. According to McCabe, the number of connected component p is equal to 1, so the final McCabe equation is formulated as:

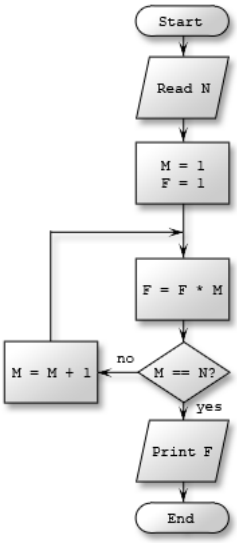

Figure 3.6: Example of CC: 8 vertices (nodes), 8 edges; $CC = 8 - 8 + 1 = 1$; image source: `http://upload.wikimedia.org/wikipedia/commons/1/16/Flowchart_example.png`.

$$v(G) = e - v + 2 \tag{3.9}$$

where e is a number of edges and v is a number of vertices. It is also another formula that gives the simple CC dependent to the number of vertices:

$$v(G) = d + 1 \tag{3.10}$$

where d is number of decision vertices points in code where the binary decision are made. This formula allows to quick determination of CC value by counting how many in code, the keywords `if, while, for, switch, case, do, catch` are used.

The low value of CC determines that given function is easy to understand. The greatest value is, the more complicated code is. What is more, the more complex code is, the more faults could appear [1].

3.5.2 Halstead complexity

The Halstead's metrics were proposed by Maurice Halstead. The main assumption is to determine a quantitative measure of the complexity directly from the operators and operands in the component. Metrics measure program component's complexity directly from source code, because the implementation or expression of the algorithm should reflect the execution independently of a specific platform. The main goal is to identify measurable software properties and its relations.

Halstead metrics are based on interpretation of the source code as a sequence of tokens and classifying them to be either operator or operand.

An operand is the part of a computer instruction which specifies what data is to be manipulated or operated on. The operator, in programming, has the same definition as in mathematics. It is a character that represents the specific action performed on the variables (addition, subtraction, multiplication, comparison, incrementation,...).

The **program length (N)** is the sum of the total number of operators and operands in the program:

$$N = N_1 + N_2 \tag{3.11}$$

The **vocabulary size (n)** is the sum of the number of unique operators and operands:

$$n = n_1 + n_2 \tag{3.12}$$

The **program volume (V)** is the contents of program information. It is measured in mathematical bits and is calculated as the program length times the 2-base logarithm of the vocabulary size (n) :

$$V = N \cdot \log_2 n \tag{3.13}$$

Halstead's volume (V) describes the size of the implementation of an algorithm. The computation of V is based on the number of operations performed and operands handled

in the algorithm. The program volume (V) is less sensitive to code layout than the lines-of-code measures[1].

The **difficulty level or error proneness (D)** of the program is proportional to the number of unique operators in the program. D is also proportional to the ratio between the total number of operands and the number of unique operands.

$$D = \frac{n_1}{2} \cdot \frac{N_2}{n_2} \tag{3.14}$$

The **program level (L)** is the inverse of the error proneness of the program. I.e. a low level program is more prone to errors than a high level program.

$$L = \frac{1}{D} \tag{3.15}$$

The **effort to implement (E)** is proportional to the volume and to the difficulty level of the program. It is also known as program understanding or elementary mental discrimination.

$$E = V \cdot D = \frac{V}{L} = \frac{n_1 N_2 N log_2 n}{2n_2} \tag{3.16}$$

The **time to implement (T)** is proportional to the effort. Empirical experiments could be used for calibrating this quantity. Halstead has found that dividing the effort by Stroud Number S gives an approximation for the time in seconds.

$$T = \frac{E}{S} = \frac{n_1 N_2 N log_2 n}{2n_2 S} \tag{3.17}$$

In 1967, psychologist John M. Stroud suggested that the human mind is capable of making a limited number of mental discrimination per second (Stroud Number), in the range of 5 to 20. The S value for software scientists is set to 18, thus:

$$T = \frac{E}{18} \tag{3.18}$$

[1]The definition of Halstead's metrics described above are not a part of complexity metrics, but size metrics, however these metrics are used as a base for metrics describes below which are categorized as complexity metrics. Using Halstead's size metrics is meaningless without Halstead's complexity metrics, so it is a reason why they have been introduced and explained in the same section.

3.5.3 Complexity metrics summary

The advantage of Halstead metrics is fact that it does not require in-depth and control flow analysis of program. They are able to predicts an effort and rate the error and estimate the time, so they are useful in scheduling projects. As a drawbacks of Halstead metrics, it could be noticed that it depends on usage of operator and operands in completed code and it has no use in predicting complexity of program at design level.

Calculating the CC value from the McCabe metric is rather easy. This metric determines also which application elements should be redesigned and reimplemented. However, on the other side, the number of edges in control flow does not give the full answers, because it does not distinguish nested and not nested loops from easy `case` instruction. It also does not take into account complicated conditions in decision vertices [8].

3.6 Object-oriented metrics

Object-oriented metrics are a response to the object-oriented programming paradigm that is not reflected in the methods of structure programming measurements.

3.6.1 Chidamber & Kemerer metrics

Chidamber & Kemerer metrics (CK metrics) is the set of metrics proposed by S.R. Chidamber and C.F Kemerer in 1994. They have explored features characteristic for object-oriented design: class complexity, inheritance, coupling and cohesion between classes.

Weighted Methods per Class (WMC) is a weighted sum of method from given class where single weight is represented as McCabe cyclomatic complexity. WMC is a sum of all coefficients of CC in given class and is expressed in formula:

$$WMC = \sum_{i=0}^{TM} CC_i \tag{3.19}$$

where TM - number of method in class, CC_i - cyclomatic complexity of i-method. Recommended WMC value should be equal to 20. The value has been determined by NASA scientists. They based its assumptions on experience in object-oriented projects [15].

Response for a Class (RFC) is the number of methods that can be invoked in response to a message in a class. In other words, all methods from given class and all methods which are invoked directly by these methods are counted. The greater RFC value is, the greater functionality and complexity is. It causes that the cost of testing and maintaining the system also increases. Such value means also more responsible for a class. The desirable value determined by NASA is 40 [15].

Depth of Inheritance Tree (DIT) is defined as a maximal number of super classes that given class inherits. The inheritance tree depth are characteristic for complex systems. It involves a wide range of related classes and methods and means higher cost of system maintenance. On the other hand, inheritance increases the reuse of code, which reduces the number of duplicate code [15].

Number of Children of Class (NOC) is defined as sum of all direct class descendants. Large NOC values indicate improper usage of inheritance mechanism and might result in difficulties in class testing and maintaining [15].

Coupling Between Objects (CBO) is defined as a number of classes that are coupled with given class using other mechanism than inheritance. The low level of class coupling indicates that class are independent and there are clear boundaries between them. It increases the abstraction of project, because it is easier to modify the code. The high level of CBO makes class more sensitive to changes and difficult to maintain,

therefore relationship between classes should be kept at as low level as possible. It reduces the complexity of the system and promotes better encapsulation [15].

Lack of Cohesion in Methods (LCOM) is type of metric that measures lack of cohesion of methods in class. There are several version of calculating LCOM.

LCOM1 is a difference between number of pairs of methods that refers to different attributes over the pair of methods referring to at least one common attribute.

$$LCOM1 = \{\, P - Q \quad if \quad P > Q0 \quad otherwise \tag{3.20}$$

LCOM2

LCOM3

LCOM4

3.6.2 Metrics for Object-Oriented Design (MOOD)

Metrics for Object-Oriented Design (MOOD) were created by Fernando Brito e Abreu in 1995. They are expressed in per cents and used to overall system evaluation. It determines the degree of the usage of mechanisms characteristic for object-oriented programming. The MOOD set is programming language independent.

MOOD measures the following object-oriented characteristics:

- polymorphism: Polymorphism Factor (PF)
- encapsulation (hermatization): Attribute Hiding Factor (AHF) and Method Hiding Factor (MHF)

- inheritance: Attribute Inheritance Factor (AIF) and Method Inheritance Factor (MIF)
- messaging: Coupling Factor (CF)

Polymorphism Factor defines the degree of classes overridden in subclasses.

$$PF = \frac{\sum_{i=1}^{TC} m_o(c_i)}{\sum_{i=1}^{TC} [m_n(c_i) \times dc(c_i)]} \tag{3.21}$$

where TC - the number of all classes, c_i - the following classes, $m_n(c_i)$ - new methods, $m_0(c_i)$ - inherited methods, $dc(c_i)$ - descendants of c_i class.

The nominator represents the number of possible different polymorphic situation and the denominator represents the maximum number of possible distinct polymorphic situation for class c_i. Polymorphism enables to link, by its own invocation, many instance of classes, so system becomes more complex and flexible. The typical PF value is between 4% and 18%. The greater value is, the more complicated inheritance hierarchy is. This situation causes increase the cost of maintenance and testing ([6], [15]).

Attribute Hiding Factor defines the degree of attributes encapsulation in class.

$$AHF = \frac{\sum_{i=1}^{TC} a_h(c_i)}{\sum_{i=1}^{TC} a_d(c_i)} \tag{3.22}$$

where TC - the number of all classes, c_i - the following classes, a_h - non-public attributes, a_d - all defined attributes in c_i class. The AHF value should be close to 100% because it is one of basic assumption of encapsulation that attributes are only accessible using methods ([6], [15]).

Method Hiding Factor defines the degree of method encapsulation in class.

$$MHF = \frac{\sum_{i=1}^{TC} m_h(c_i)}{\sum_{i=1}^{TC} m_d(c_i)} \tag{3.23}$$

where TC - the number of all classes, c_i - the following classes, m_h - non-public methods, m_d - all defined methods in c_i class. The desirable value of MHF should be between 10% and 25%, because some of methods are not available at all ([6], [15]).

Attribute Inheritance Factor defines the degree of attributes inheritance usage. It is a ratio of inherited and all available attributes.

$$AIF = \frac{\sum_{i=1}^{TC} a_i(c_i)}{\sum_{i=1}^{TC} a_a(c_a)} \tag{3.24}$$

where TC - number of all classes, $a_a(c_i) = a_d(c_i) + ai(ci)$ - public attributes in c_i class, $a_i(c_i)$ - inherited and not overridden attributes in c_i class, $a_d(c_i) = a_n(c_i) + a_o(c_i)$ - attributes defined in c_i class, where $a_n(c_i)$ - new attributes, $a_o(c_i)$ - inherited and overridden attributes. The typical value of AIF is between 50% and 60% ([6], [15]).

Method Inheritance Factor defines the degree of methods inheritance usage. It is a ratio of inherited and all available methods.

$$MIF = \frac{\sum_{i=1}^{TC} m_i(c_i)}{\sum_{i=1}^{TC} m_a(c_a)} \tag{3.25}$$

where TC - number of all classes, $m_a(c_i) = m_d(c_i) + m_i(c_i)$ - public methods in c_i class, $m_i(c_i)$ - inherited and not overridden methods in c_i class, $m_d(c_i) = m_n(c_i) + m_o(c_i)$ - methods defined in c_i class, where $m_n(c_i)$ - new methods, $m_o(c_i)$ - inherited and overridden methods. The typical MIF value is between 65% and 80%. The values below indicates weak usage of inheritance, otherwise, the values greater than 80% complicates the inheritance and code re-usage ([6], [15]).

Coupling Factor defines the degree of coupling between classes using different methods than inheritance. It is a ratio of number of couplings between classes with use of aggregation, composition, association and maximal number of couplings that could be used in system.

$$CF = \frac{\sum_{i=1}^{TC} \left[\sum_{j=1}^{TC} is\_client\,(c_i, c_j)\right]}{TC^2 - TC + 2 \times \sum_{i=1}^{TC} dc(c_i)} \tag{3.26}$$

where TC - number of all classes, $dc(c_i)$ - descendants of c_i class, $TC^2 - TC$ - maximal number of possible couplings in TC class system, $2 \times \sum_{i=1}^{TC} dc(c_i)$ - maximal number of coupling using inheritance in system, $is\_client\,(c_i, c_j) = 1 \quad if \quad c_c \Rightarrow c_s \wedge c_c \neq c_s \wedge \neg\,(c_c \rightarrow c_s)$ otherwise $is\_client\,(c_i, c_j) = 0$. The $c_c \Rightarrow c_s$ expression means that class c_c

has at least one reference to class c_s. The $c_c \rightarrow c_s$ expression means that class c_c inherits class c_s. The CF value is between 5% and 15%. The high value means that system is not flexible and is difficult to maintain and test, because one change in the system requires modifications in another parts of system. Small value means that classes are too complex, not cohered and it realizes too much functionality inside ([6], [15]).

3.7 Package metrics

Package metrics are used to measure software at the package level. They are used to evaluate the correctness of coupling. The most popular set of package metrics were introduced by Robert C. Martin.

3.7.1 Martin metrics

Martin metrics are set of five metrics designed by Robert Cecil Martin in 1994. His metrics *"can be used to measure the quality of an object-oriented design in terms of the interdependence between the subsystems of that design, because designs which are highly interdependent tend to be rigid, unreusable and hard to maintain"* [14].

Efferent Coupling (Ce) is also known as *Outgoing Dependencies*. This metric measures all the types from the source of the measured package referring to the types not in the measured package [14]. A large Ce indicates that a package is unfocussed and unstable, because it depends on the stability of all the types to which it is coupled. The recommended value of Ce is upper limit of 20. The Ce could be reduced by extracting some parts of classes and decomposing into smaller and more specialized classes. The typical example of large valued efferent coupling are GUI elements which covers all logical operation within a software.

Afferent Coupling (Ca) determines how many classes and interfaces from other packages depend on classes in analysed package. It is also known as *Incoming Dependencies*. The number of packages that depend on the analysed package indicates the package level of responsibility. If the package is relatively abstract then a large number of incoming dependencies is acceptable. Otherwise if not, the package is more concrete. The acceptable values are much larger than in the case of Ce. The allowed value for Ca is about 500. It is caused by difficulty of the control of the packages which depend on the analysed package. An example of Ca high value is controller from MVC pattern [14].

Instability (I) is measured by calculating the effort necessary to change a package without impacting other packages within the application. Instability is also called *Stable Dependencies Principle*. The number of incoming and outgoing dependencies is an indicator that determines the stability and instability of a package. Instability can be calculated by comparing the incoming and outgoing package dependencies:

$$I = \frac{Ce}{Ca + Ce} \tag{3.27}$$

The returned value is in the range between 0.0 and 1.0 where 0.0 indicates a maximally stable package and 1.0 indicates a maximally unstable package. However, it is impossible to clearly state about stability of package, so it have been assumed that package is stable within range: 0.0 to 0.3 and unstable from 0.7 to 1.0.

Summing up, the package containing multiple outgoing but few incoming dependencies is less stable because of the consequences after introducing changes. On the other hand, package containing more incoming dependencies are more stable because it is more difficult to change [14].

Abstractness (A) calculate a relationship between number of abstract classes and interfaces within package and total number of concrete (non-abstract) classes in the same package. The abstractness is calculated using the formula:

$$A = \frac{N_a}{N_c} \tag{3.28}$$

where N_a is a number of abstract classes and interfaces in a package and N_c is a number of concrete classes and interfaces in the package. The abstractness value of zero ($A = 0$) indicates that package contains complete concrete classes. The abstractness value of one ($A = 1$) indicates that package is completely abstract.

Having abstractness value in range between 0 and 0.5 determines that package is opened to changes of implementation. However it could be stated so, whether this value will be compared to instability of package. If package is highly unstable, it would be hard to change because of stability and such package cannot be extended because it is not abstract.

If the abstractness value is closed to 1, the package is unstable and it should consist of concrete classes. It is need to limit dependences to unstable types. Using abstract classes as unstable types will cause increased of dependences to them, because, in case of abstract classes, it is need to create class that will inherit from abstract class [14].

Distance from the Main Sequence (D) points that abstractness and stability of packages are closely connected. This metrics is expressed using the formula:

$$D = A + I - 1 \tag{3.29}$$

The ideal value is $D = 0$, what means that the more abstract a package is, the more stable it should be because it should have many clients that depend on its abstractions.The desirable value should as low as possible [14]. Any package that is not closed to zero is considered as unbalanced and should be reimplemented in order to be more reusable and less sensitive to changes.

3.8 Other software metrics

There are also many other software metrics that are commonly used to preventing the faults in software. They increase the quality of code and have been implemented in plug-ins in Eclipse IDE described in Chapter 4. The alphabetic list presents short description of each of them:

Comment Lines of Code (CLOC) counts all lines that contain regular comments and Javadoc comments.

Cyclic Dependencies (CYC) estimates how many cycles in which a package is involved.

Density of Comments (DC) determines a density value of how commented the code is and is expressed in formula: $DC = \frac{CLOC}{LOC}$.

Dependency Inversion Principle (DIP) calculates the ratio of dependencies that have abstract classes or interfaces as a target.

Direct Cyclic Dependencies (DCYC) counts every mutual dependency between packages.

Encapsulation Principle (EP) calculates the ratio of classes that are used outside of a package.

Executable Statements (EXEC) counts the number of executable statements.

Limited Size Principle (LSP) is the number of direct subpackages of a package.

Non-Comment Lines of Code (NCLOC) (aka NCSL and ELOC) counts all the lines that do not contain comments or blank lines.

Number of Abstract Types (NOTa) counts the number of abstract classes and interfaces.

Number of Concrete Types (NOTc) counts the number of concrete classes.

Number of Exported Types (NOTe) counts the number of classes and interfaces exported outside a package.

Number of Parameters (NP) counts the number of parameters for a method or a constructor.

Number of Types (NOT) counts the number of classes and interfaces.

Number Of Fields (NOF) calculates the number of fields declared in method.

Number Of Attributes (NOA) calculates the number of fields declared in class or interface.

3.9 Summary

Chapter 4

Tools implementing software metrics

This chapter describes the tools (plugins) for Eclipse IDE that implements software metrics.

4.1 Stan - Structure Analysis for Java

[opis]

4.2 RefractorIT

[opis]

4.3 Sonar

[opis]

4.4 JDepend

[opis]

4.5 CKJM - Chidamber and Kemerer Metrics

[opis]

4.6 Simian - Similarity analyzer

[opis]

4.7 Cobertura

[opis]

4.8 Findbugs

[opis]

4.9 Checkstyle

[opis]

Chapter 5

Practical approach to software metrics

Chapter 6

Conclusions

Bibliography

Books

[1] Abran A., *Software Metrics and Software Metrology*, IEEE Computer Society, Los Alamitos, 2010.

[2] Berge C., *Theory of Graphs*, Dover Publications, 2001.

[3] Booch G., *Object-Oriented Analysis and Design with Applications*, Addison-Wesley, 2007.

[4] Fenton N., Pfleeger S., *Software Metrics - A Rigorous And Practical Approach*, PWS Publishing Company, Boston, 1997.

[5] Kan S., *Metrics and Models in Software Quality Engineering*, Addison Wesley, 2002.

[6] Sarker M., *An overview of Object Oriented Design Metrics*, Department of Computer Science at Sweden Umea University, Umea, 2005.

[7] Sierra K., Bates B., *Sun Certified Programmer for Java 6*, Mc Graw Hill, 2008.

Online documents

[8] Aivosto.com, *Complexity metrics*, [access: November 9, 2013].
Available at: `http://www.aivosto.com/project/help/pm-complexity.html`

[9] Algorithms Overview, *A directed graphs*, [access: November 9, 2013]. Available at: `http://algs4.cs.princeton.edu/42directed/`

[10] Cornett S., *Code Coverage Analysis*, [access: November 3, 2013]. Available at: `http://www.bullseye.com/coverage.html`

[11] ISO 9126 Software Quality Characteristics,*An overview of the ISO 9126-1 software quality model definition, with an explanation of the major characteristics*, [access: November 26, 2013]. Available at: `http://www.sqa.net/iso9126.html`

[12] Java Tutorials, The, *Polymorphism*, [access: October 7, 2013]. Available at: `http://docs.oracle.com/javase/tutorial/java/IandI/polymorphism.html`

[13] Johnson P., *Testing and Code Coverage*, [access: November 3, 2013]. Available at: `http://pjcj.net/testing_and_code_coverage/paper.html`

[14] Martin R. *OO Design Quality Metrics. An Analysis of Dependencies*, [access: November 11, 2013]. Available at: `http://www.cin.ufpe.br/~alt/mestrado/oodmetrc.pdf`

[15] Rosenberg L., *Applying and Interpreting Object Oriented Metrics*, Software Assurance Technology Center (SATC) in NASA, [access: November 9, 2013]. Available at `http://www.literateprogramming.com/ooapply.pdf`.

List of Figures

List of Tables

List of Acronyms

A Abstractness

AHF Attribute Hiding Factor

AIF Attribute Inheritance Factor

Ca Afferent Coupling

CAD Computer Aided Design

CBO Coupling Between Objects

CC Cyclomatic Complexity

Ce Efferent Coupling

CF Coupling Factor

CK metrics Chidamber & Kemerer metrics

CLOC Comment Lines of Code

CYC Cyclic Dependencies

D Distance from the Main Sequence

DC Density of Comments

DCYC Direct Cyclic Dependencies

DIP Dependency Inversion Principle

DIT Depth of Inheritance Tree

EP Encapsulation Principle

EXEC Executable Statements

I Instability

IDE Integrated Development Environment

LCOM Lack of Cohesion in Methods

LLoC Logical Lines of Code

LoC Lines of Code

LSP Limited Size Principle

MHF Method Hiding Factor

MIF Method Inheritance Factor

MOOD Metrics for Object-Oriented Design

MQ Modularization Quality

NCLOC Non-Comment Lines of Code

NOA Number Of Attributes

NOC Number of Children of Class

NOF Number Of Fields

NOT Number of Types

NOTa Number of Abstract Types

NOTc Number of Concrete Types

NOTe Number of Exported Types

NP Number of Parameters

PF Polymorphism Factor

RFC Response for a Class

SLoC Source Lines of Code

WMC Weighted Methods per Class