Software metrics are units of software measurement. They are used to characterize software products, development process, as well as software engineers. Over the years, software metrics have been a useful tool to control the development and maintenance activities of software applications.

If metrics are used properly, they allow gaining multiple benefits. Firstly, metrics help make quantified and meaningful estimates. Secondly, they support a decision making process from business to technical level. Thirdly, they identify and quantify products and process improvements, examine engineers efficiency and productivity. Lastly, based on metrics it is possible to define success and failure (or degree of success and failure) for a product or a process [BER03].

Traditional metrics have been applied to the measurement of structured systems since 1976 [MCC76]. But the object-oriented paradigm has made great contribution to the study of software measures. The multiplicity of interrelated elements of software and the increased awareness of quality assurance throughout the object oriented system development led to the development of appropriate metrics.

These object-oriented metrics have two issues that need to be addressed: measuring and the resultant measure, and the object-oriented paradigm.
5.1. **OBJECT-ORIENTED METRICS AND MEASURES**

Traditional software metrics such as lines of code or cyclomatic complexity have been used in industry as indicators of quality since 1976 [MCC76]. Nevertheless, as object-oriented approach emerged to support major applications, the effectiveness of applying traditional software metrics to object-oriented systems was challenged. The following issues were presented as the main reasons for the inadequacy of using traditional metrics to object-oriented software [MOR90]:

- assumptions relating program size and software engineers productivity in structured systems do not apply directly to object-oriented software,
- traditional metrics do not address the design aspects of object-oriented systems,
- the computation of the system's complexity as the sum of the complexity of each components is not appropriate for large object-oriented systems.

On the other hand other works indicate that traditional metrics are applicable to the measurement of the complexity of object-oriented systems and can be used to quantify software complexity [TEG92].

The object-oriented design approach gives opportunity to classify metrics naturally. The classification captures object-oriented software features and properties hierarchically. It begins with the high-level characteristics of an object-oriented system and moves down to the low-level characteristics.

**Source code size metrics**

Traditional metrics which are applied to object oriented software give insight into an overall system size and allow comparing systems and evaluating productivity. They can also be used as a refactoring effectiveness indicator.

Lines of Code (LOC) metric is most common software project measure. The metric becomes a baseline to measure the degree of work performed on a project and it is used to create time and cost estimates [ENC10].
Effective Lines of Code Metric (eLOC) is a measure of all lines that are not comments, blanks or standalone braces or parenthesis. This metric more closely represents the quantity of work performed [RES10].

Comment Line and Comment Percent (or Comment to Code Ratio) is a degree of commenting within the source code. It measures the care taken by programmers to make the source code and algorithms understandable. Poorly commented code makes the maintenance activities an extremely expensive. Recommended minimum is 20%.

Blank Line and White Space Percent Metric is the number of blank lines within source code. It indicates the readability of product.

And File Count Metric counts the files processed and generates metrics based on the file extension. It provides the distribution of the source code types, source code types and distribution of the specifications to the implementations [RES10].

**Procedural metrics**

Cyclomatic Complexity is a popular procedural (called also function) software metric equal to the number of decisions that can be taken in a procedure [MCC76]. A decision is defined as an occurrence of keywords such as: "while", "for", "foreach", "continue", "if", "case", "goto", "try" and "catch" within the function. Cyclomatic Complexity is the sum of these constructs.

That metric helps to identify software need of inspection or redesign, and also to allocate resources for evaluation and test [SAS10]. It suggests that module approach decreases defects when McCabe's Cyclomatic Complexity is within 2 and 7 [EBE97].

Another interesting procedural metrics are Fan In and Fan Out. Those values measure the relationships between files and procedures and indicates the complexity of the static structure of code.

**Class metrics**

Class metrics describe structure of a class and relationship between classes. The volume of a class is a basic size measure connected with the amount of information inside it. The class volume can be measured by Number of Variables and by Number of Methods. Also Average LOC per Class and per Method metrics can provide insight
into the average module size in the system. The methods where LOC is higher than 20 are hard to understand or maintain and should be refactored.

The internal structure of an object is based on the methods and is an indicator of its functionality. Method metrics are used to estimate effort for testing early. Those metrics can be measured by Number of Parameters per Method, Weighted Methods per Class, Maximum Nesting Level, Method Rank.

Number of Parameter per Method counts parameters of a method and also references. Methods where metric is higher than 5 might be difficult to call and could decrease the performance. It is recommended to provide a class dedicated to handle arguments passing.

Weighted Methods per Class metric is a sum of complexities of methods defined in a class. It represents the complexity of a class as a whole and can be used to indicate the development and maintenance effort for a class.

Maximum of Nesting Level for a method is the maximum number of encapsulated scopes inside the body of the method. Methods where this metric is higher than 4 are hard to understand and maintain [NDE10, SHE98].

Method Rank metric is computed using Google Page Rank algorithm on the graph of methods dependencies. That metrics should be tested carefully, because the metric errors can be detrimental.

Classes often interact with other classes to form a subsystem. Excessive coupling between objects is harmful to a module design and it reuse [ARC95a]. Number of Invoked Classes is one of the metrics that can be used to identify such references [EBE97]. Afferent Coupling and Efferent Coupling at method level are another object coupling metrics.

Afferent Coupling for a particular method is the number of methods that depends directly on it and the Efferent Coupling for a particular method is the number of methods it directly depends on. Afferent Coupling is an indicator for the responsibility. The higher this value is the higher is the element’s responsibility. Efferent Coupling means that a element depends on several other implementation details and it makes it instable. Therefore it is good practice to keep the Efferent Coupling for all artefacts at a minimum.
The inheritance relationships characteristic between classes and their parents indicate to a designer where changes would improve the development. The metrics connected to classes inheritance should take into account both the depth and breadth of the relationships [ARC95b]. The Height of Inheritance Tree metric is counted as the maximum number of nodes from the class node to the root of the inheritance hierarchy. The deeper within the hierarchy, the more methods the class can inherit, increasing its complexity.

**Product metrics**

The measurable characteristics of a system as a whole might be presented as a number of classes, a number of unites test, a percentage of code coverage by tests [PUR03].

Another approach to product metrics represent Product Metric of Halstead. In that metric software is measured by counting the number of operator and operands, and for them the program vocabulary, length and volume is calculated. Halstead’s metric has several advantages. Firstly, it do not require in-depth analysis of programming structure, so is simple and can be used for any programming language. Secondly, it allow to predict an error rate. And lastly, using that metric is possible to determine effort and time needed to develop software [SAS10].

### 5.2. Impact of Metrics on Software Development Life Cycle

Software metrics can be applied throughout the whole software development process. There are different usage goal and different set of metrics to evaluate each of the phases of object-oriented software development life cycle, such as requirements and analysis, design, coding, testing and maintenance.

During requirements and analysis phase, the metrics that measure completeness and specificity of the system requirements are used. The metrics like specificity of requirements, completeness of functional requirements and degree to which the requirements have been validated are proposed [SAS10].
The metrics such as McCabe’s Cyclomatic Complexity, Information Flow (fan in/out), Depth of Inheritance Tree, Maximum Nesting Level, Number of Class methods are applied at design phase. They are helpful for defining test cases, allocating resources for evaluation and test, identifying needs for inspection or redesign. Those metrics are used on different design decomposition levels.

The software metrics appropriate during implementation are Lines of Code or various defect metrics. They are taking into consideration software quality, developer’s performance, a need of code refactoring or review, dependencies of components and so on.

For software testing and maintaining, the focus should be placed on metrics that help to follow up the number of defects in various system aspects and to predict the reliability of software products. Recommended metrics are: Reliability Growth Logarithmic Model [SAS10], Number of Test Case, Number of Unit Test, Code Coverage, etc [LAN10].

5.3. **Analysis of PHP Project**

Presented analysis was conducted for the CMS project for the financial organization. The CMS was implemented using PHP5 framework called Symfony and ORM framework Doctrine. The system is consistent with the object-oriented paradigm.

The analysis based on the results from two programs: Understand by SciTools and PHP_Depend [PHP10, SCI10]. The files such as CSS, JS and images was omitted during the analysis.

The source code metrics, generated using Understand, program are presented in tab. 5.1. The project consists 1916 classes with 12 174 methods. The Comment to Code Ratio was calculated at 6%, thereby exceeding the average value [EBE97].

Table 5.1. Source size code metrics for CMS project generated by SciTools Understand

<table>
<thead>
<tr>
<th>Metric name</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blank Lines</td>
<td>39 098</td>
</tr>
<tr>
<td>Classes</td>
<td>1 916</td>
</tr>
</tbody>
</table>
The code distribution is also presented in the chart (fig. 5.1).

The average cyclomatic complexity for all project nested functions or methods is 2 with the maximum equals 85. The value does not exceed suggested maximum of average complexity [EBE97]. On the other hand, the project maximum of nesting is higher than recommended [NDE10]. The visual demonstration of the complexity results is presented in fig. 5.2.
Table 5.2. Cyclomatic complexity generated by SciTools Understand

<table>
<thead>
<tr>
<th>Metric name</th>
<th>AvgCyclomatic</th>
<th>MaxCyclomatic</th>
<th>MaxNesting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result</td>
<td>2.06</td>
<td>85</td>
<td>6</td>
</tr>
</tbody>
</table>

The fig. 5.3 presents the base structure of the Overview Pyramid [LAN10, MCC76], which is used to visualize a complete software system in a legible manner, generated by another program – PHP_Depend. The pyramid collects a set of metrics from the categories such as inheritance, coupling, size and complexity, and their relations.

The results obtained from PHP_Depend are similar to the results from Understand. The number of class within the project is calculated at 1832 with around 1200 methods and 155 000 lines of code. The Cyclomatic Complexity (presented as CYCLO) equals also 2.
Average Lines of Code per Method (as LOC to NOM ratio) is almost 17. The value of the metric does not cross recommended maximum of 20 code lines per a method, but is high and implicates possible difficulty in understanding the methods’ mechanism.

Calculated Number of Method (presented as NOM to NOC ratio) is around 7. The value is within the range. but it can be explained by high Average LOC per Method metric value. Thus, in that project class readability has been preserved at the expense of methods simplicity. It possible that after refactoring this indicator will change.

The Overview Pyramid presents also metrics informing about software inheritance and coupling. The results for two inheritance metrics, Average Number of Derived Classes (ANDC) describes the average of derived classes and Average Depth in Inheritance Tree (AHH), are awarded. They both exceed maximum value shown in reference table (tab. 5.3). Thus it means that the inheritance structure in presented program is too complex and can cause the problems with it understanding.

In contrast, coupling metric counts as a number of distinct method-calls per a number of method (presented as CALLS) is average. That can implicate the adequate level of methods specialization.

The Information Flow metric (presented as FANOUT to CALL ratio), which is three times lower than minimum, and low number of class per package (NOM to NOP ratio) suggests low program complexity.
Table 5.3. Reference values for Overview Pyramid

<table>
<thead>
<tr>
<th>Metric</th>
<th>Low</th>
<th>Average</th>
<th>High</th>
</tr>
</thead>
<tbody>
<tr>
<td>CYCLO/LOC</td>
<td>0.16</td>
<td>0.20</td>
<td>0.24</td>
</tr>
<tr>
<td>LOC/NOM</td>
<td>7</td>
<td>10</td>
<td>13</td>
</tr>
<tr>
<td>NOM/NOC</td>
<td>4</td>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>NOC/NOP</td>
<td>6</td>
<td>17</td>
<td>26</td>
</tr>
<tr>
<td>CALLS/NOM</td>
<td>2.01</td>
<td>2.62</td>
<td>3.2</td>
</tr>
<tr>
<td>FANOUT/CALLS</td>
<td>0.56</td>
<td>0.62</td>
<td>0.68</td>
</tr>
<tr>
<td>ANDC</td>
<td>0.25</td>
<td>0.41</td>
<td>0.57</td>
</tr>
<tr>
<td>AHH</td>
<td>0.09</td>
<td>0.21</td>
<td>0.32</td>
</tr>
</tbody>
</table>

The AI chart (fig. 5.4) was created from the calculated values to visualize the relationship between the instability on the y axis and the abstraction on the x axis.

The elements presented in the chart represent the program packages. Their location close to the diagonal means good balance between the package abstraction and instability. Because most of packages are distant to the diagonal, the conclusion is that program is inappropriate balanced and should be redesigned.
Table 5.4 The selected metrics for class pageComponents

<table>
<thead>
<tr>
<th>Class metric name</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Afferent Coupling</td>
<td>0</td>
</tr>
<tr>
<td>Coupling Between Objects</td>
<td>11</td>
</tr>
<tr>
<td>Efferent Coupling</td>
<td>11</td>
</tr>
<tr>
<td>Comment Lines to Code</td>
<td>13</td>
</tr>
<tr>
<td>Code Rank</td>
<td>0,15</td>
</tr>
<tr>
<td>Depth of Inheritance Tree</td>
<td>2</td>
</tr>
<tr>
<td>Lines Of Code</td>
<td>211</td>
</tr>
<tr>
<td>Number Of Added Methods</td>
<td>12</td>
</tr>
<tr>
<td>Number Of Child Classes</td>
<td>0</td>
</tr>
<tr>
<td>Weighted Method Count</td>
<td>28</td>
</tr>
</tbody>
</table>

The selected metrics are presented in tab. 5.4 for the example class called pageComponents. The class contains 211 lines of code with 13 lines of comments. It is the second in the inheritance tree and no class inherits for it. All of 12 class methods are added – none of methods is overwritten. High value of efferent Coupling metric means that the class depends on 11 other implementation and it makes the class instable.

5.4. TOOL INFORMATION

*PHP_Depend* is a small program that performs static code analysis on a given source base. Static code analysis means that *PHP_Depend* first takes the source code and parses it into an easily processable internal data structure called an *Abstract Syntax Tree*, that represents the different statements and elements used in the analyzed source base [PHP10].

*Understand* is a powerful, easy-to-use software tools developed by Scientific Toolworks Inc. *Understand* is a set of sophisticated static analysis tools that help measure and maintain software source code. In contrast to *PHP_Depend, Understand*
is a commercial program that supports many programming languages such as C/C++/C#, Ada, Java, FORTRAN, Delphi and Jovial programming languages [SCI10].

5.5. CONCLUSIONS

There are many traditional and object-oriented metrics useful for an object-oriented software measure. Those metrics can be simply distinguish into two categories by applying construction rules [EBE97]:

- the static and dynamic structure of a class or an object,
- the static and dynamic relationships between classes and objects.

Metrics for object-oriented software are used to characterize products, software engineering processes and software developers. And collecting metrics in running projects helps to:

- assess the quality of code,
- focus on code reusability and maintainability,
- determine the improvement or degradation in a project,
- build up a historical database for future estimations and risk assessments,
- verify a software design or architecture [LOR94].

But it is extremely important to note that metrics are almost always interrelated – a change of one metric has impact on other metrics for the same product or process. What is more, to be useful metrics should be collected regularly and in a structured way. Otherwise, they can be detrimental.

Finally, it must be mentioned that the most useful set of metric for specific product, process or organization is not defined. The set of metrics should be work out after analysis of various software engineering aspects and collected set of metrics.
REFERENCES


[ARC95b] Archer C.: Measuring Object-Oriented Software Products (http://www.sei.cmu.edu/reports/95cm028.pdf), 1995


[PHP10] PHP Depend (http://pdepend.org/), 17.12.2010

n.htm), 10.11.2010


