MECOT: A Software Quality Metrics Collection Tool

Yusuf Umaru Mshelia
yumshell@gmail.com

DOI: 10.20470/jsi.v10i1.360

Abstract: Numerous software quality metric tools have been developed in response to the growing need in software quality evaluation. Most of these tools, however, are developed to the constraints of system requirements, platform restrictions and/or organizational definitions as there exist many (more than 1000) quality metrics that have tools for a selected metrics. More specifically, the restricted flexibility of modifying existing tool to achieve a comprehensive metrics repository, the creation of multiple quality assessment environments and unrelated result collection platform in the analysis of metric results are the problems identified. Hence, this research developed a web-enabled software metrics collection tool by integrating existing Open Source Software (OSS) metric tools as components to the new platform independent environment—the web. The objectives were tailored to i) integrate selected software metrics ii) improve the flexibility of modifying existing tools for enhanced extensibility, and iii) To develop a metric collection tool with least platform dependence with focus on selected traditional metrics. MEtrics COllection TOol (MECOT) - an applet, is the resulting software product that is designed to collect new metrics as class-based components.

Key words: Software metrics; software quality; metrics collection tool; MECOT;

1 Introduction

In order to design good quality software that meets quality models, high quality software, are targeted from requirement gathering to design, implementation, testing and maintenance. Software Testing and quality assurance is a process maintained from the early stage of requirement and analysis stage that ripples down to implementation where the program source code is generated (Pressman, 2005).

Because of software code generation, testing and quality assurance is an important part of the software engineering professional practice that lasts the evolutionary life-cycle of the software. This quantitative evaluation is targeted on the source code. Source-code attribute measures in quality assurance is important in determining further modifications, extensions, maintainability of the software, possibly the software entropy (extinctions) and similar source code-determinant features of the software quality.

In determining the degree of quality of software, specific desirable qualities are expected at a threshold level. Essential qualities like maintainability, usability, reusability, reliability, etc., are discretely measured and the quality established on the make-up attributes of the source code like Lines of Code (LOC), Cyclomatic Complexity (CC), Depth of Inheritance (DIT), Number of Error Messages, Fan-in/fan-out, Depth of Conditional Nesting, etc.

A quantitative measure of the degree to which software items possess a given quality attribute is the quality metric of that software (Radatz et al., 1990). It is computationally a function whose inputs are software data (usually the source code) and whose output is a single numerical value that can be interpreted as the degree to which the software possesses a given quality attribute. (Radatz et al., 1990). Consequently, traditional software quality metrics is the equivalent quantifiable measure of software attributes that are fundamental to structured design without account of object orientation. Hence, software metrics can measure quality by manually computing and examining the source code or using their equivalent automated tools.

Given a said metric function, an automated software tool that takes in a source code as its input, to produce, through a user interface the numerical output which the software evaluates as a quality attribute is the Software Quality Metrics Tool. It is hence clear that software metrics tools are fundamentally useful in software testing and quality assurance.
1.1 Problem Statement
For the many (thousands) existing software metrics (Fenton and Neil, 2000), there is first a manually computational process or at best an automated quality metrics tool for a subset of the many metrics. These many different existing software metric tools are mostly re-engineered to extend or used as plugins and in some cases completely reengineered for extensibility. Hence, the following deduced specific problems this research addressed:

1. Restricted flexibility of modifying existing tool to achieve a comprehensive metrics repository
2. The creation of multiple assessment environments and unrelated result collection platform to analyze metric result.

1.2 Aim and Objectives
This research aimed at developing a web-enabled software metrics collection tool by integrating existing Open Source Software (OSS) metric tools as components to the new platform environment - the web. Hence, the objectives of this research are identified as follows:

1. To develop a metric collection tool with least platform dependence (web based) with focus on traditional metrics.
2. To integrate selected software metric tools as class-based components.
3. To improve the extensibility metrics tools.

Section I highlights software quality, metrics and the automated tools. Literature and tools with the implementing technology is discussed in II. Section III Justifies the relevance of this work and IV reported the methodology. In V, VI and VII, the results, conclusion and recommendations are made.

2 Software Quality, Metrics and Tools
Any engineering approach (including software engineering) rests on an organizational commitment to quality. Total quality management, Six Sigma, and similar philosophies foster a continuous process improvement culture, and it is this culture that ultimately leads to the development of increasingly more effective approaches to software engineering. The bedrock that supports software engineering is a focus on quality (Pressman, 2005). Following up on achieving quality in software development is therefore done through quality assurance.

When a planned and systematic pattern of all actions necessary to provide adequate confidence of an item or product conforms to established technical requirement, the process is quality assurance (Radatz et al., 1990). This is however practicable against a measurement standard. This is the main instrument in the continual monitoring of the software product throughout the development process. Software metrics is a collective term used to describe the wide range of activities concerned with measurement in software engineering. These activities range from producing numbers that characterize properties of software code (Fenton and Neil, 2000). Software metrics has been categorized majorly into two: product and process metrics (though with other exceptions that consider project metrics or resource metrics which is here considered as part of process metric (Sommerville, 2004).

Product metrics reflect characteristics of software product itself. Some software features are visible to the user, while others are only visible to a development team. In accordance to availability of observed attributes to participants in development process and end user, software metrics can be divided into internal and external. This is the line of metrics this work intends to explore tools for. Process metrics (Project metrics inclusive), however, are related to the characteristics of the process. They are used with intention to improve the observed software development process. Frequently used measure is the amount of effort needed to invest in the entire software development in specific phases or in specific activities during the development phase. These efforts are reflected by some of the standard measures such as man-months or man-days. Other examples of the process metrics are frequency of repetition of errors, the time required to eliminate or reduce errors, etc. Project metrics represent certain values related to resources and costs, their allocation to the individual stages in the evolution of products, productivity, and other items related to project planning and management (Gordana, 2015).

The history of software metrics is traced back to the mid-1960s when the Lines of Code (LOC) metric was introduced as basis for measuring programming productivity and effort (Fenton and Neil, 2000). Product size measure (as in LOC/KLOC for thousands of LOC) is assumed to drive any predictive
model. Soon, obvious drawbacks of dwelling on such a crude measure as LOC was insufficient as a surrogate measure for such different notions of program size were recognized in the mid-1970s. The need for more discriminating measures became necessary with the increasing diversity of programming languages since an assembly language is not comparable in effort, functionality, or complexity to a LOC in a high-level language. There soon grew a rapid interest in 1970s in software complexity as pioneered by (Halstead, 1977) and (McCabe, 1976). With unfolding demands in the software community and the shift from procedural to object-oriented programming, some more metrics like Morris Metrics (Morris, 1989), C&K Metrics (Chidamber and Kemerer, 1994) and MOOD (e Abreu, 1995). Lorenz and Kidd also expounded the theme in their work on software metrics (Lorenz and Kidd, 1994).

Since then, many new metrics are initiated and adapted today to meet the technological trends in the evolution of software, quality and the predictive requirements in software engineering processes. As a result, (Fenton and Neil, 2000) reported that there are literally thousands of metrics that have been proposed since the mid-1960s as was maintainability metrics alone recorded to be above 5000 (de AG Saraiva et al., 2015).

2.1 Lines of Code (LOC)

Lines of code (LOC), also known as source lines of code (SLOC), is a software metric used to measure the size of a computer program by counting the number of lines in the text of the program's source code. SLOC is typically used to predict the amount of effort that will be required to develop a program, as well as to estimate programming productivity or maintainability once the software is produced.

Consider this snippet of C code as an example of the ambiguity encountered when determining SLOC (source lines of code):

```c
} /* count the lines of code*/
for (i = 0; i < 100; i++)
{
    printf("hello");
}
```

In this example, we have 4 SLOC, 1 comment (C) line (CLOC) and 5 SLOC+C or (C&SLOC). Other forms of lines of code are Logical Lines of Code (LLOC or SLOC-L), Physical Lines of Code (PLOC or (LOC-P), Block Lines of Code (BLOC).

2.2 Software metrics tool

Software engineering tools provide automated or semi-automated support for software processes and the methods of their development. When tools are integrated so that information created by one tool can be used by another, a system for the support of software development, called computer-aided software engineering CASE, is established (Sommerville, 2004).

3 Literature Review

This section reviews research works that have been done in the development of software quality metrics tools (under Related tools on software quality metrics) and the attempts of integrating software metrics and their corresponding automated tools (in Related works on metric tools' comparison).

3.1 Related tools on software quality metrics

Attempts in automating software metrics is traced back to ATHENA software certification tool (Christodoulakis et al., 1989).

Focus was quickly shifted to metric tools for the analysis of software development projects and the trend was propelled with the introduction of PROMIS (Kokol et al., 1995) with a model framework for generating metrics tool at development life-cycle stage. This was a result of supporting software metrics use which were becoming extremely important for understanding software behaviour and its design and use process. This was when conventional software analyzers had several limitations and were complex computer programs not easy to design, implement and maintain.
Crocodile (Lewerentz and Simon, 1998) provided static analysis features that can be integrated into an existing development environment. The goal of the project Crocodile is to provide concepts and tools for an effective usage of quantitative product measurement to support and facilitate design and code reviews. The application field is the realm of object-oriented programs and, particularly, reusable frameworks. Its main concepts are measurement tool integration into existing software development environments, using existing tool sets and integration mechanisms, to define flexible product quality models based on a factor-criteria-metrics approach, the use of meaningful measurement contexts isolating or combining product components to be reviewed, filtering and presentation of measurement data. This tool particularly provides these concepts as TakeFive’s SNIFF+, an industrial strength integrated C++/Java programming environment (Pfeiffer, 1997). Crocodile provides static analysis features e.g. in Audit-C/C++ by SemaGroup (Marinescu, 1997) and it can be fully integrated into an existing development environment.

Sun (now Oracle) introduced JMetrics (Farnese et al., 1999), a Java code analysis tool that implements analysis of Java source codes only as a stand-alone application.

MOOSE, a tool environment (Buhler, 2003) also came with a reverse re-engineering approach to software measurement based on the FAMIX model. The solution to result presentation problem to the user is a tool that displays the computed measurements using a graphical user interface. Moose uses MooseGager, a tool that displays the computed measurements of the entities of the underlying model in a simple way and also offers the possibility to generate charts based on these measurements. These and other features of this tool provide an interface to the Moose reengineering environment that helps the user to use the available measurements.

PROM (Sillitti et al., 2003) and its sub-tool WebMetrics (Scotto et al., 2004) proposed an abstraction layer and an architecture supporting metrics collection and analysis. From the recognition of shortage of automated tools for collecting and analyzing measures, the researchers added that it does not contribute to the evolution of software reengineering. It also has as part of its constituents a personal software process (PSP) data. The tool uses an architecture based on plug-ins that automatically collects data from development tools.

OOMeter (Alghamdi et al., 2005) computes metrics on source code as well as XMI files and is extensible by Java classes that implement a certain interface. However, this runs only on Java platforms and works on Java and C# source code or XMI file generated from UML case tool. OOMeter is envisioned as a tool that can be used to quantitatively measure several quality attributes of each to the artifacts produced during the lifecycle of a software development project. These artifacts include requirements specifications, design models, source codes and test specifications. At the moment, however, only source code and design level metrics are supported. This also is a stand-alone tool on the build.

QMetrics (Schackmann et al., 2009) with its Bugzilla component provides a general infrastructure for specifying metrics, relating them to organization-specific quality models, and automatic evaluation based on empirical comparison data. It also provides flexible tool support to define and evaluate quality models based on software metrics. It implements metrics appropriate for organization-specific information needs must be implemented in custom scripts. Hence developing and validating new metrics is time-consuming and costly. Furthermore, available tools lack a flexible approach on how to model the relation of metrics to higher-level goals. Its strength is on Change Request (CR), which make provision for organization-specific metric.

FLOSSMetrics (Herraiz et al., 2009) constructs, publishes and analyses a large-scale database with information and metrics about libre software development. The main objective of FLOSSMetrics was to construct, publish and analyse a large-scale database with information and metrics about libre software development coming from several thousands of software projects, using existing methodologies, and tools already developed. The project also provides a public platform for validation and industrial exploitation of results.

PREST (Kocaguneli et al., 2009) collects source code metrics and call graphs in five different programming languages with a deep-learning based defect prediction and analysis capability. It is an open-source stand-alone Java tool that that collects source code metrics and calls graph in 5 different programming languages and performs its computations.

Another tool is the open source, stand-alone Squale (Bergel et al., 2009, Mordal-Manet et al., 2009) which reportedly implements software metrics for all input languages. The Squale project was born from industrial effort to control software quality. Its goals are to refine and enhance Qualixo Model, a software-metric based quality model already used by some companies in France and to support the estimation of return on investment produced by software quality. Qualixo Model is a software quality
model based on the aggregation of software metrics into higher level indicators called practices, criteria and factors. The main goals of Squale are to evaluate and enhance the existing software quality approach to evaluate and enhance the current software-metric, quality model, including current practices and current metrics; define dashboards related to software quality; enable assessment evolution of software quality, provide economic indicators to assess added value (ROI) of software quality measurements; and disseminate acquired knowledge through an open-source platform supporting the model and a community of users. The SQUALE project aimed at building an anonymous centralized database where audit results will be stored.

Other recent tools that have gained attention in the software community are Goanna (Vogelsang et al., 2010), a user-definable metrics, an implementation and integration in the existing static analysis, QUAMOCO tool chain (Deissenboeck et al., 2011) which integrates code analysis tools with quality models. MASU (Higo et al., 2011) unifies independent languages and analyzes its intermediate representations with a pluggable extensibility feature. SSQSA framework (Gordana, 2015) also focuses on language independency for metrics implementations with similar tools like the SQUALE. Many more commercial, freeware and open source tools exist with the motivation to estimate and analyze software quality. A comprehensive listing of the tools examined in this research is available in the meta-analysis Table in Appendix A.

These surveyed tools show how essential software metrics tools are in research and how many works are product specific needs and within restrained by certain development environment that may not other environments.

From these listings, many can be identified as open source and others commercial, even though some started from the research environment before commercializing. Other Open source tools worthy of mention which have no roots in any published work are srcML, SDMetrics, Xogastan, GCC, coqua, JDepend, splmetrics, devAdvantage, devCodeMetrics, metricsAnalyzer, metrixplusplus, LOCMetrics, source monitor, CCCc, codeAnalyzer, Metrics eclipse plugin, CKJM, etc.

In addition, some other studies also compared existing metrics tools based on functionalities and what metrics they measure. This studies first established in the experimentation with some commercial and free metrics tool by Lincke (2008). The research showed that existing software metric tools interpret and implement the definitions of object-oriented software metrics differently. The focus was directed at how metrics-based assessment of software systems for the purpose of improving design which showed a noticeably different output from tool to tool. The work proved different outputs in the data generated in the different tools experimented. They reported on the subset of the number of metrics the tools calculated and compared the results. The tools’ implementation capability with respect to suitability, usability and compatibility with and across different platforms was not their focus. Hence, a consequent gap in holistic implementation capability of tools across different systems.

Another work was by Novak & Rakić (2010) with much similarity to Lincke et al.’s but for the focus on metrics tools for the .NET environment. The work further presented the statistical analysis of difference in output using the T-Test. However, since the analysis is restricted to the .NET platform, platform-dependent characteristics for dissimilar execution environment may not apply to .NET. Hence, unable to report the reality in that analyzed data for a different platform. Furthermore, the work does not also cover a study on the implementation details.

Jain et al in (Jain, et al., 2014) attempted a theoretical comparison of some tools with the goal of integrating the functionalities of java coded and command-line executable programs. As in (Novak & Rakić, 2010), the study is constrained to Java programmed tools. Similarly, tool’s execution environment was not the focus as in (Lincke, et al., 2008) and (Novak & Rakić, 2010).

Some studies on metrics tools compared existing metrics tools based on functionalities and what metrics they measure in experimentation with some commercial and free metrics tool (Mshelia et al., 2017).

Furthermore, all three works have selected a limited number of metrics which do not portray the full capabilities of the tools evaluated. For example, whereas CCCC evaluates up to eight (8) metrics, (Lincke, et al., 2008) captured four (4). Similarly, (Novak & Rakić, 2010) captured six (6) metrics of NDepend whereas it can measure up to 82 metrics. Thoroughly, the study did not capture the full power of the tools. In understanding the uniqueness, differences and similarities of software metrics tools, this work considers a holistic study to be able to answer these abstracted questions. Hence, a work that complements the existing work with special focus on execution details and system characteristics to jointly put forth a holistic projection of the differences and/or homogeneity between the capabilities of these tools and their implementation descriptions. These works are all focused on the functionalities of the tools. This work however focuses of the tools’ execution details; platforms of
execution, execution requirement and the system-centered restrictions, consequences and applications.

The findings of Mshelia et al. (2016) confirmed that for different software metrics tools as adapted in Project L3, the relationship in the output values computed by the tools vary between a positively strong and negative correlation to negatively strong and weak correlation to perfectly positive and negative correlations. In addition, there exists no correlation at all to some degree for all the project classes for which results are attempted to be correlated. Another work studied software metric tools with a focus on the implementation requirements to demonstrate the symmetry and variations of these tools for a holistic extensibility of software metrics in industrial and research environments. Fifteen (15) characteristics of metric tools based on execution and capability were studied and a clear distinct ratio of 3:2 in the average uniform to non-uniform result pattern is obtained (Mshelia et al., 2017).

3.2 Tools and Technology
The tool selected for reengineering is JavaLOC. Other tools used during the development of MECOT are:

1. JD GUI 1.4 – A Java decompiler that converts java bytecodes to source code
2. Eclipse Neon 3 – An Integrated Development Environment (IDE) for developing applications
3. jdk1.8.0_131 – Java Development Kit

A laboratory is set up on the windows 8 system from which all research work held.

4 Justification and Significance
Determining the quality of software from their corresponding source code attributes is an important quality assurance and control mechanism, hence, a need for a uniform result assessment platform tool that can automate the equivalent manual process.

The omissions in the automation of static code analysis of relevant software metrics for the improvement of the existing tools through an integrated software metrics collection tool is the basis for the proposition of this tool-MECOT (MEtrics COllection Tool). In this attempt to systematically apply the use of quantitative empirical analysis in software quality analysis, an appropriate tool for metrics calculation constitutes important steps towards the success of software projects universally.

Preliminary review and investigation of the state of the art in the field shows that there is no wider acceptance in real life product quality monitoring (Gordana, 2015). The main problem in wider application of software metrics as affirmed by the researcher lays in the limitations and inappropriateness of involved tools. The different tools often give inconsistent results for the same metric algorithm and support only a few selections of applicable metrics. This is because the rule for metrics calculations could be differently interpreted with different tools (Scotto et al., 2004).

This research attempts through MECOT to have an entity tool for which all independent functional components can be collected with much more support to bring software metrics application. This is justifiable by reducing platform differences and having a central implementation and result collection platform.

Furthermore, if these many metrics tools must run on a single computer system, the weight of resources the tools will demand can highly reduce the system performance. Therefore, a platform independent tool which runs, say on a web browser will encourage and enhance acceptance and in effect, static code analysis by empirical evidence.

5 Methodology

5.1 Analysis and Design Stage
This work used the criteria below in selecting and qualifying the metrics tools used in the design of MECOT.

1. Open-source development design (OSD)
2. Component-based software design (CSD) approach

These two approaches formed the framework for implementing the next stage and is demonstrated in a work flow chart described in Figure 1.
5.2 Implementation Stage
From the surveyed software, those that met the criteria for the above were collected for integration with minimal modification. The Software Reengineering process model is adapted in the design. This idea is shown in Figure 2. This model explains the stages involved in modifying software within an evolutionary context.
The design of MECOT took the path of the architecture in Figure 2. The applet view makes use of class-based components that is interfaced to the applet that implementing on the architecture of Figure 3 over a 12-months period.

6 Results

6.1 Design Stage
At completion, MECOT has a functional interface which measures LOC, LOC+C and CLOC. The interfaces are in the Figures 4-8.
MECOT: A SOFTWARE QUALITY METRICS COLLECTION TOOL

Figure 4: MECOT Page 1, source: (author)

Figure 5: MECOT Choose Files, source: (author)

Figure 6: MECOT java file selection window, source: (author)
7 Conclusion

This work further established that software metrics tools have different implementations that make the launching of the tools inconsistent with different platforms in concurrent executions. Research on the execution differences in the effect and ease of an integrated environment that accepts multi-faceted programs has been established to be a challenge in integrating, collecting and analyzing quality attributes on a central system.

This research attempted through MECOT to have an entity tool for which all independent functional components can be collected with more support to bring software metrics into applications based on empirical values for quality assessment in software development with three (3) variants of size metrics (LOC) as the start-up metrics in MECOT. MECOT is a futuristic tool with focus on extensibility against a "fixed" tool. It is projected to add to the open source community where developers can add class-based components.

8 Recommendation and future work

Since implementation of metric algorithm has been yielding inconsistent result, standardization of implementation should be defined from an acceptable reference point in the input source code. With increasing metric tools, attempt to answer to what extent this observed difference will be of concern to the universal interpretation of software metrics output has received little attention? If the result of metrics tools will continually be relatively interpreted, standards are at risks of indifference with respect to interpreting and application of metrics result as same output level may be considered acceptable.
and by another, failure. If attention is not paid, unguided proliferation in standard will explode with increased number of metric tools. This will uncontrollably increase self-defined standards and tools.

Being that MECOT is built on java applet technology, it is still platform dependent. A web-based metrics tool is suggested in future work. MECOT may not contain the complexity of high increase in the number of components to be integrated.

9 References


## 10 Appendix A

<table>
<thead>
<tr>
<th>Tool</th>
<th>Author</th>
<th>Title</th>
<th>Metrics</th>
<th>Research interest/Method/focus</th>
<th>Strength</th>
<th>Gaps</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>OOMeter</td>
<td>(Alghamdi et al., 2005)</td>
<td>OOMeter: A Software Quality Assurance Tool</td>
<td>1. Ratio of Cohesive Interactions (RCI) 2. Weighted Method per Class (WMC) 3. Depth of an Inheritance Tree (DIT) 4. Number of Children (NOC) 5. Coupling btw Objects (CBO) 6. Response for a class (RFC) 7. Lack of Cohesion on Methods (LCOM) 8. Tight Class Cohesion (TCC) 9. Loose Class Cohesion 10. Lines of Code (LOC)</td>
<td>Measure quality attribute of each artefact (source code &amp; design level metrics,) produced during a software development project and works on Java and C# source code or XMI file generated from a UML case tool</td>
<td>Accepts Java &amp; C# Interface for users to define custom metrics through java classes that implement a certain interface</td>
<td>Limited to XML file generated from a UML case tool</td>
<td>1. Not Traditional Software Metrics 2. Stand-alone</td>
</tr>
<tr>
<td>MASU</td>
<td>(Higo et al., 2011)</td>
<td>A Pluggable Tool for Measuring Software Metrics from Source Code</td>
<td>The C&amp;K Metrics (WMC, DIT, NOC, CBO, RFC, LCOM, CC)</td>
<td>Metrics measurement platform MASU: Multi language Metric tool based on C&amp;K Metrics</td>
<td>Independent of other parametric tools</td>
<td>Measures only 7 metrics</td>
<td></td>
</tr>
<tr>
<td>PREST</td>
<td>(Kocaguneli et al., 2009)</td>
<td>Prest: An Intelligent Software Metrics Extraction, Analysis and Defect Prediction tool</td>
<td>Static Code Metrics Predictor &amp; analyzer</td>
<td>Use of Machine Learning methods for analysis and defect prediction</td>
<td>Ability to define new metrics Outputs are in 5 formats thru GUI component Use of Machine Learning methods</td>
<td>Limited to OO static metrics Limited to OO Metrics, Java based standalone</td>
<td>Relevant to my work</td>
</tr>
<tr>
<td>Tool</td>
<td>Research References</td>
<td>Description</td>
<td>Benefits</td>
<td>Notes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>--------------------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------</td>
<td>-------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QMetrics</td>
<td>(Schackmann et al., 2009)</td>
<td>QMetric - A Metric Tool Suite for the Evaluation of Software Process Data</td>
<td>Specifies metrics with automatic evaluation based on empirical comparison data</td>
<td>General infrastructure to evaluate metrics on s/w process data</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>QMetric-bugzilla Metrics</td>
<td></td>
<td>Flexible tool support to define and evaluate quality models based on software metrics.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Goanna</td>
<td>(Vogelsang et al., 2010)</td>
<td>Software Metrics in Static Program Analysis</td>
<td>GOANNA-visualization approach for the display of metrics results</td>
<td>User-defined metrics to user-defined visualization</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>GOANNA-</td>
<td></td>
<td>For Static OO metrics</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Project size metrics (19), class size metrics (7), method size metrics (6), Project Characteristics Metrics (7), Detection of flaw (NOP overrides, NOP overrides, large classes, long methods, misplaced methods)</td>
<td>User-defined visualization Metric specification Language (GMSL)</td>
<td>Commercially licensed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOOSE</td>
<td>(Buhler, 2003)</td>
<td>MooseGager, a Software Metrics Tool based on Moose</td>
<td>Reverse engineering and re-engineering OO systems environ.</td>
<td>Does not cater for new metrics</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Project size metrics (19), class size metrics (7), method size metrics (6), Project Characteristics Metrics (7), Detection of flaw (NOP overrides, NOP overrides, large classes, long methods, misplaced methods)</td>
<td></td>
<td>Not for traditional s/w metrics</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLOSSMetrics</td>
<td>(Herraiz et al., 2009)</td>
<td>FLOSSMetrics: Free / Libre / Open Source Software Metrics</td>
<td>Mining tools integrated: CVSAnalY, Mailing List Stats, Bicho, Libresco's CMetrics package, SLOCCount, CCC, PyMetrics, PerlMetrics</td>
<td>The use of existing methodologies, and tools already developed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>construct, publish and analyze a large-scale database with information and metrics about libre software development</td>
<td>Focus on metric repository</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PROMIS</td>
<td>(Kokol et al., 1995)</td>
<td>PROMIS: a software metrics tool generator</td>
<td>A general architecture for generating new metrics tool</td>
<td>Focus on tool generation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tool</td>
<td>(Authors, Year)</td>
<td>Description</td>
<td>Language(s) Evaluated</td>
<td>Integration</td>
<td>Personal Software Process (PSP) Focus</td>
<td>Tool Type</td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>----------------</td>
<td>-------------</td>
<td>-----------------------</td>
<td>------------</td>
<td>------------------------------------</td>
<td>-----------</td>
<td></td>
</tr>
<tr>
<td>PROM</td>
<td>(Sillitti et al., 2003)</td>
<td>Collecting, Integrating and Analyzing Software Metrics and Personal Software Process Data</td>
<td>Personal software process (PSP) data, focus on project effort</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQUALE</td>
<td>(Bergel et al., 2009)</td>
<td>SQUALE – Software QUALity Enhancement</td>
<td>Open source</td>
<td></td>
<td></td>
<td>Stand-alone</td>
<td></td>
</tr>
<tr>
<td>QUAMOCO</td>
<td>(Deissenboeck et al., 2011)</td>
<td>The Quamoco Tool Chain for Quality Modelling and Assessment</td>
<td>Focus on quality assurance &amp; modelling</td>
<td>Quality models ranging from abstract characteristics down to operationalized measures</td>
<td>Works on dependent tool kit</td>
<td>Stand alone</td>
<td></td>
</tr>
<tr>
<td>SSQSA</td>
<td>(Gordana, 2015)</td>
<td>Extendable and Adaptable Framework for Input Language Independent Static Analysis</td>
<td>Framework on the meta-structure of programming languages</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**JEL Classification: C88**