Automated COSMIC Measurement of Java Swing Applications throughout their Development Life Cycle

Nadia Chamkha[1], Asma Sellami[1], and Alain Abran[2]

[1] Mir@cl Laboratory, University of Sfax, Tunisia.
nadia.chamkha@gmail.com, asma.sellami@isims.usf.tn
[2] Department of Software Engineering and I.T., ETS –University of Quebec, Canada
alain.abran@etsmtl.ca

Abstract. In highly competitive organizations, measurement is crucial to control projects to meet customer requirements. Despite some successes in software, there are few mechanisms for the developers to rapidly and objectively verify and track the amount of functionality to be delivered. COSMIC – ISO 19761 functional size measurement can be used to keep track of software through its development life cycle. In this paper, we propose a "JavaCFP" plugin tool for measuring the COSMIC functional size of java source code being developed. This JavaCFP tool can be used for controlling the completeness of implemented functionality against specified requirements, for identifying deviations and for generating progress reports on the implementation of new functions. JavaCFP was developed in NetBeans IDE and the "C-REG" case study is used as an example to illustrate this plugin.

Keywords: COSMIC method, Functional size measurement, Java swing application, Automated measurement, Plugin, ISO 19761.

1 Introduction

A crucial task for software project management is to evaluate what percent of the promised functionality has been completed or, while the code is being written, how much functionality remains to be developed. A timely and objective evaluation is a key factor for tracking project progress, decision making, monitoring, and so on. A project might appear to be successful if it is within the 'approved' budget but, without the ability to verify that all the promised functions have been delivered, there is always a possibility that only a portion of the corresponding promised functions has indeed been delivered for the budget approved for the full set of functions: therefore, when less functions are delivered within the initially estimated budget, it is then improper to claim that the budget estimates were correct [1]. Many software projects have been canceled after large investments of effort, time, and money because of both inadequate initial estimating and no objective determination of the status of the work products leading to a credible re-estimation of a completion date or the cost to complete the project for its approved scope [2].

Thus, software developers need a rapid mean to objectively measure the amount of work done. Such a measurement mean would provide all stakeholders (developers, managers, suppliers, customers, etc.) with an objective basis to monitor and control the completeness of functional requirements.

For the measurement of functional requirements, many researchers have proposed to automate the COSMIC Function Points (ISO 19761) method for the sake of less human involvement, such as in [3] [4] [5] [6], and high accuracy for the measurement tool proposed by Soubra [7] [8] in the context of software requirements documented in a specification tool format, such as in [9] [10] [11] [12] [13] [14] [15] [16] [17] [18]. However, these studies did not tackle the on-going monitoring of projects throughout the development lifecycle.

This paper proposes a tool for sizing java swing applications while they are being developed, and its use throughout the software development life cycle. This tool is mainly based on the COSMIC measurement process. The remaining of this paper is structured as follows. Section 2 presents an overview of the COSMIC method and some related works on COSMIC automation. Section 3 presents the COSMIC measurement process for sizing Java Swing applications. Section 4 illustrates our measurement through the case study "C-REG" source code. Section 5 illustrates the implementation of "JavaCFP" with a comparison among tools based on metrology concepts. Finally, section 6 presents some conclusions and suggestions for further works.

2 Background

2.1 Overview of the COSMIC Method

The COSMIC – ISO 19761 method offers a standard way of sizing the functional user requirements (FUR) of all types of software developed in any type of environment. It can be applied in any of the software life-cycle phases, from the requirements to their implementation in code. Basically, COSMIC measures the software functional size from the FUR representing the "user practices and procedures that the software must perform" as mandated by ISO 14143 [19]. COSMIC defines a three-phase process for measuring the functional size of software: the measurement strategy phase, the mapping phase, and the measurement phase [20].

- The measurement Strategy Phase: it includes the identification of a number of measurement context parameters to ensure that the measurement results can be correctly interpreted in the future. These parameters involve the Purpose of measurement and the Scope of the software to be measured. The output of the strategy phase is the "Software Context Model" including the identification of the software Layers, their Functional users, the level of granularity of the documentation and the identification of the Persistent storage of the piece of software to be measured.
- The mapping phase: in this phase, the FUR are mapped to the COSMIC "Generic Software Model". Each FUR involves a number of functional processes (FP) where each consists of a set of functional sub-processes that move data or manipulate data. For instance, a data movement moves a single data group from/to a user (re-

spectively Entry and eXit data movement) or from/to a persistent storage (respectively Read and Write data movement).
- The Measurement phase: In this ISO standard, each data movement of a single data group is assigned a measurement unit of 1 CFP (COSMIC Function Point). The software functional size is computed by adding all data movements identified for every functional process.

2.2 Related work on COSMIC automation with Java

The automation of COSMIC method has gained interest in managing software projects: automation can help practitioners to measure objectively and quickly the size of their software through the software life-cycle, from early development phase [9] [3] [4] [10] [5] [12] [7] [6] to the late coding phase ([13], [14], [15], [16], [18]). The focus of this related work section is on FSM automation in the coding phase. For instance:

- Akca et al. [13] proposed a semi-automated functional size measurement of the source code in "three_tier java business application" using COSMIC version 3.0.1. This measurement is achieved by the use of a self-developed "measurement library" of functional processes that are triggered via the GUI of the application. The measurement results led to 92% accuracy as compared to the manual measurement.
- Akca et al. [21] proposed to compare the costs of semi-automatic and manual measurements of three case studies. The results showed that the automatic process can reduce measurement costs by up to 280% compared to the manual measurement when integrated at the beginning of the coding phase.
- Sag et al. [14] [15] proposed a COSMIC measurement tool ('Cosmic Solver') of source or binary code. They proposed a number of rules to derive UML Sequence Diagrams from the software execution at runtime with AspectJ technology. 'Cosmic Solver' was demonstrated using a three-tier Java business application sample. The functional size extracted by the prototype was 96,8% convergent to the one obtained by manual measurement.
- Gonultas et al. [16] proposed to automate COSMIC for GUI Web Java business applications based on a three-tier architecture. Like [13], the automation was limited to developing a "Measurement Library" requiring the installation of Java application code. The library can be used only for applications with a specific architecture that use technologies of JSF, Spring and Hibernate. The automated measurement converged by 94% to the manual measurement and also reduced measurement duration by about 97% (e.g., 1/34 of manual measurement effort). Small accuracy deviations were related to the technology and to the parsing method.

Table 1 presents a summary of the tools in these related studies, including the authors, the automation context and the automation tool.

Table 1. Automation Tools for COSMIC and Java Language in the Coding Phase

Authors	Context	Tool
Akca et al. [13] [21]	Business application at run-time, software based on three-tier architecture in Java	Run-time measurement, Semi-automated Measurement Library by being imported and by making small code additions
Sag, M.A et al. [14] [15]	Three-tier Java business application at run-time for java source/object code	Automated measurement by the "Cosmic Solver" tool
Gonultas et al. [16]	Run-time Java business applications and web-based GUI with a specific architecture using JSF, Spring and Hibernate	Run-time measurement Automated Measurement Measurement Library for Java Application by "Static Code Installer" Component

3 COSMIC Process for Java Swing Applications

This section presents the COSMIC automated measurement process designed for sizing Java Swing applications.

3.1 The measurement strategy phase

- Purpose of measurement: to measure precisely the size of Java Swing applications being implemented in NetBeans through the proposed JavaCFP tool.
- Measurement Scope: all the functionality allocated to software as specified in the java source code, i.e. all files having the extension « .java ».
- Functional user: two main types are identified (e.g., external user and software/system components). An external user can be an individual (e.g., user of the application being measured) who interacts manually and directly with the java swing application to be measured. An external software/system component (e.g., another program such as the proposed tool "JavaCFP", services, etc.) is in a direct relation with the application being measured. Note that the automation tool user (e.g., developers) interacts with the application being measured only via the JavaCFP.
- Level of granularity: the level of the invoked methods implemented in a listener interface (e.g., event handling code). The Level of the GUI without its associated Documentation (FUR) does not allow the usage of the COSMIC.

3.2 The mapping phase

- Triggering events: in java source code, the event listener interfaces (e.g., `ActionListener`, `MouseListener`, etc.) attached to a "Swing control" (e.g., `JButton`, `JTextField`, etc.) written in a general way as `<source_object>.add<Evt_type>Listener(<Listener_object>)` can be identified as the triggering events, where:

  - `Source_object` is a "Swing Control"
  - `Evt_type ≡ Xxx` is an event type attached to "Swing Control" (`Action, Item, Text, Menu` etc.)
  - `Listner_object` is an instance of the class implementing `XxxListener`
- Functional process: corresponds to the invoked event-handler or the invoked callback method (e.g. `NameOfMethodActionPerformed(), NameOf-MethodWindowClosing()`). Once an event occurs for which there is a listener, the source event calls the method which is provided in the listener. Such a method is known as a callback method or event-handler method written as follows:
  `public void <callback_method> (<Evt_type>Event event) { <invoked_callback_method>(event); }`
- Object of interest: An object of interest could not have only one corresponding object in Java. Assuming that each event listener interface is associated to an invoked callback method (i.e. one FP), the object of interest will correspond to the java object class. Else, it will correspond to a set of attributes derived from two or more object classes. In Java Swing the object-class is declared as public class:
  `public class <NameOfObject> extends Object`
- Data Group: corresponds to the data fields (`TextField, List, Password-Field, radioButton,` etc.) describing the same object of interest.
- Data attribute: corresponds to each data field of each object class (such as simple object, inheritance, composition, aggregation) or derived attributes from two or more object classes.

Table 2. The Mapping Rules "Java Swing Application"/COSMIC

#	**Rule**
RP1	All java source codes having the extension ".java" and placed in the "src" folder should correspond to all FUR.
RP2	The body of Method corresponding to java instructions (detail of each called procedure) refers to the level of decomposition.
RP3	NetBeans platform corresponds to the layers according to the scope of the code to be measured.
RP4	The interface between functional users and the Java application refers to the boundary.
RP5	Each invoked callback_method that is the method triggered by a java event corresponds to a FP.
RP6	Level of the invoked methods implemented in a "listener interface" (e.g., ActionListner, etc.) refers to the level of granularity.
RP7	Event listener interfaces (e.g., ActionListener, MouseListener, etc.) attached to a "Swing control" (e.g., JButton, JTextField, etc.) corresponds to the triggering event
RP8	Java object class or the set of derived attributes from more than one object classes corresponds to an object of interest.
RP9	Data fields corresponding to the attributes of either simple Object or a set of attributes derived from two or more object classes (such as simple object, inheritance, composition, aggregation) are referred to as data group.
RP10	Each data field of each object-class or derived attributes corresponds to a data attribute.

The alignment of the COSMIC concepts with those of java swing applications as described above is crucial for identifying a set of mapping rules (RP1 to RP10) in Table 2. These rules could be applied in general and used to generate the measurement rules (RM1 to RM 4) in Table 3 and (RM5 and RM6) in Table 4, and their corresponding measurement formulae.

3.3 The Measurement Phase

Table 3 presents the rules (RM1 to RM4) for identifying the data movement types. For each data movement the value of 1 CFP is assigned. The rules (RM5 and RM6) are used to provide the functional size of each functional process and that of the source code as a whole (Table 4).

Table 3. Rules for Identifying Each Data Movement

#	**Rule**
RM1	Each java method such as `getText(), getSelectedItem(), showInputDialog(), getValueAt()` should be classified as **Entry** data movement when triggered by the same event listener.
RM2	Each java method such as `setText(), showOptionDialog(), showInternalMessageDialog(), showInternalOptionDialog(),showInternalConfirmDialog(), showMessageDialog(), NotifyDescriptor.Message(), addItem(), Append(), System.out.println()` should be classified as **eXit** data movement and triggered by the same event listener.
RM3	Each java syntax `[SELECT…executeQuery]` should be classified as **Read** data movement and triggered by the same event listener.
RM4	Each java syntax such as `INSERT…executeUpdate, delete from … executeUpdate, update… executeUpdate` should be classified as **Write** data movement and triggered by the same event listener.

Table 4. Measurement Rules for determining the functional size of the swing application and each functional process derived from the source code.

#	**Rule**
RM5	The FS of the java Source code (or GUI with its FUR documentation) is equal to the sum of the sizes of its functional processes (FP).
RM6	For each Invoked Callback_Method (i.e. FP), the FS of the Invoked Callback_Method is equal to the number of its data movements.

Based on the measurement rules RM1 to RM6, we propose the following set of measurement formulae (1) to (7) associated with java concepts for the design of the "JavaCFP" tool.

$$FS(SwingApplication) = \sum_{i=1}^{n} FS(Invoked\_Callback\_Method_i) \quad (1)$$

Where:

- *FS(Swing Application)*: Functional size of the Java Swing Application or GUI.
- *N*: number of methods triggered by an event (i.e. number of *invoked_Callback_Method*) in the Java Swing application.

$$FS(Invoked\_Callback\_Method_i) = \sum_{j=1}^{4} x_{ij} \quad (2)$$

- *FS(Invoked_Callback_Method$_i$)*: Functional size of the *Invoked_Callback_Method$_i$* written in java language

$$- \; j = 1 \iff E \; in \; Method \; i \; ; \; x_{i1} = n_1 \times E \quad \forall x_{i1} \in \begin{cases} getSelectedItem()_i \\ showInputDialog_i \\ showInternalInputDialog_i \\ getValueAt_i \\ getXxx_i \end{cases} \quad (3)$$

$$where \; n1 = \begin{cases} number \; of \; non \; similar \; Entries \\ 1 \; otherwise \end{cases}$$

$$- \; j = 2 \iff X \; in \; Method \; i \; ; \; x_{i2} = n_2 \times X \quad \forall x_{i2} \in \begin{cases} showOptionDialog_i \\ showInternalMessageDialog_i \\ showInternalOptionDialog_i \\ showMessageDialog_i \\ NotifyDescriptor.Message_i \\ addItem_i \\ setXxx_i \\ System.out.println_i \end{cases} \quad (4)$$

$$where \; n2 = \begin{cases} number \; of \; non \; similar \; \text{eXits} \\ 1 \; otherwise \end{cases}$$

$$- \; j = 3 \iff R \; in \; Method \; i \; ; \; x_{i3} = n_3 \times R \quad \forall x_{i3} = [SELECT \ldots executeQuery_i \,] \quad (5)$$

$$where \; n3 = \; number \; of \; reads \; (\text{n3} >= 0)$$

$$- \; j = 4 \iff W \; in \; Method \; i; x_{i4} = n_4 \times W \quad \forall x_{i4} \in \begin{cases} [insert \; into \; \ldots executeUpdate_i] \\ [delete \; from \ldots executeUpdate_i] \\ [update \ldots executeUpdate_i] \end{cases} \quad (6)$$

$$where \; n4 = \; number \; of \; writes \; (\text{n4} >= 0)$$

$$- \; \forall j \in \{1,2,3,4\} \;\; if\big(x_{ij} = \emptyset\big) \; then \; n_i = 0 \quad (7)$$

4 Illustrative Example: Execution of the Proposed Formulae

To illustrate the application of these proposed measurement formulae and show how measurement results can be provided, we use the Course Registration ('C-REG' V2.0) System Case Study [22]:

- First, we implemented the FUR (Maintain Student Data) of "C-REG" in NetBeans. Due to space limitation, we present here only the GUI of the Functional Process "Add Student's details" (Fig. 1) and its corresponding piece of code (Fig. 2) in which the extracted data movement types are presented.
- Second, we present in details how to measure the Functional Size of the functional process (FP) "Add Student's details" manually from the code.

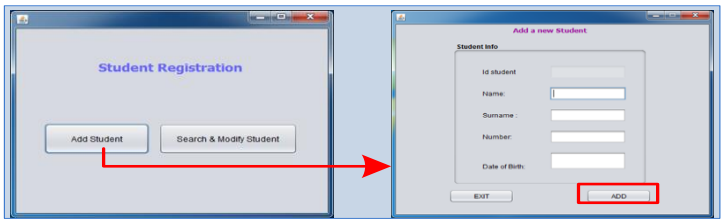

Fig. 1. GUI to add a new student

```
private void Add_StudentActionPerformed(java.awt.event.ActionEvent evt) {
    Student S = new Student();
    String cle=null;
    S.setName_Student(jTextField1.getText());
    S.setSurName_Student(jTextField2.getText());
    S.setNumber_Student(Integer.parseInt(jTextField3.getText()));
    S.setDate_birth_Student(jTextField5.getText());
    connexion o = new connexion();
    cle=o.Add_Student(S);
    jTextField6.setText(cle);
}

    public String Add_Student(Student S) {
        connectBase();
        Statement stmt;
        String selectens;
        ResultSet rs=null;
        String cle=null;
        try {
            stmt = connexion.createStatement();
                selectens="INSERT INTO student "
                        + "(Name_Student, SurName_Student, Number_Student)"
                        + "VALUES('"+S.getName_Student()+"','"+S.getSurName
            int i=  stmt.executeUpdate(selectens,stmt.RETURN_GENERATED_KEYS)
            if (i > 0) {
                JOptionPane.showMessageDialog(null, "Success Insertion");
                rs=stmt.getGeneratedKeys();
                rs.next();
                cle=rs.getString(1);}
            }
        catch (SQLException e) {
                JOptionPane.showMessageDialog(null, "Error Insertion\n" + e.
                }
        return cle;
    }
```

1E
1X
1W
1X

Fig. 2. Source code associate to « ADD » button

By applying formula (2), the functional size of the FUR 'Add student's details is equal to 4 CFP - see below formula (2) for FS("ADD Student's details"). This measurement result is the same as provided in [22].

$$FS(Add\_StudentActionPerformed()) = \sum_{j=1}^{4} x_{1j}$$
$$= x_{11} + x_{12} + x_{13} + x_{14}$$
$$= n_1 \times E + n_2 \times X + n_3 \times R + n_4 \times W$$
$$= 1E + 2X + 0R + 1W$$
$$= 4\ CFP$$

$n_1 = 1$ (four similar functions `getText()`)
$n_2 = 2$ (one `setText()` and two similar functions `showMessageDilog()`)
$n_3 = 0$ (no function associated to Read data movement)
$n_4 = 1$ (one function insert `into…executeUpdate`)

5 Sizing Automatically Java Swing Application

5.1 Implementation

This section illustrates the implementation of "JavaCFP" with a comparison among tools based on metrology concepts. This tool is called "JavaCFP" which is a NetBeans plugin that implements our proposed measurement formulae (section 3.3). By using this tool, software developers will be able to generate:

- The functional size of Java Swing applications being written and after their development in the NetBeans environment (e.g., the functional sizes of each Entry, eXit, Read and Write of any java Method triggered by an event).
- The functional size of Java Swing applications after making a functional change (i.e., the added/modified method).
- Useful information to verify if the implemented functionality meets the COSMIC rules. Examples:
  - detection of missing Entry data movement in a functional process.;
  - information on the development progress reflecting the development of a new/modified functional process.

To illustrate how our "JavaCFP" plug-in tool works, we use as an example the piece of code associated with the "Maintain Student Data" in "C-REG" case study [22]. Fig. 3 shows the screen where the GUI functional size is generated. Note that the measurement results provided by "JavaCFP" tool are the same as those provided manually in [22].

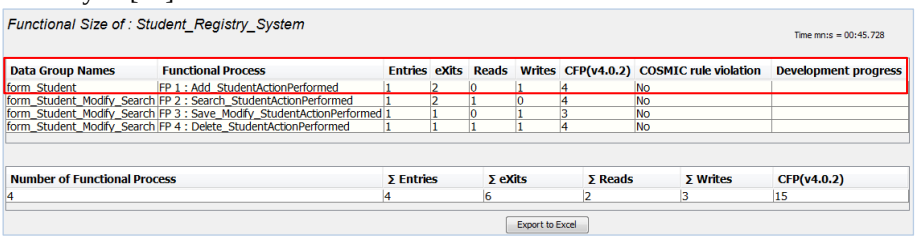

Fig. 3. Sizing the "Maintain Student Data" with "JavaCFP" tool

Fig. 4 gives some snapshots of the "JavaCFP" tool. Only four interfaces are presented.

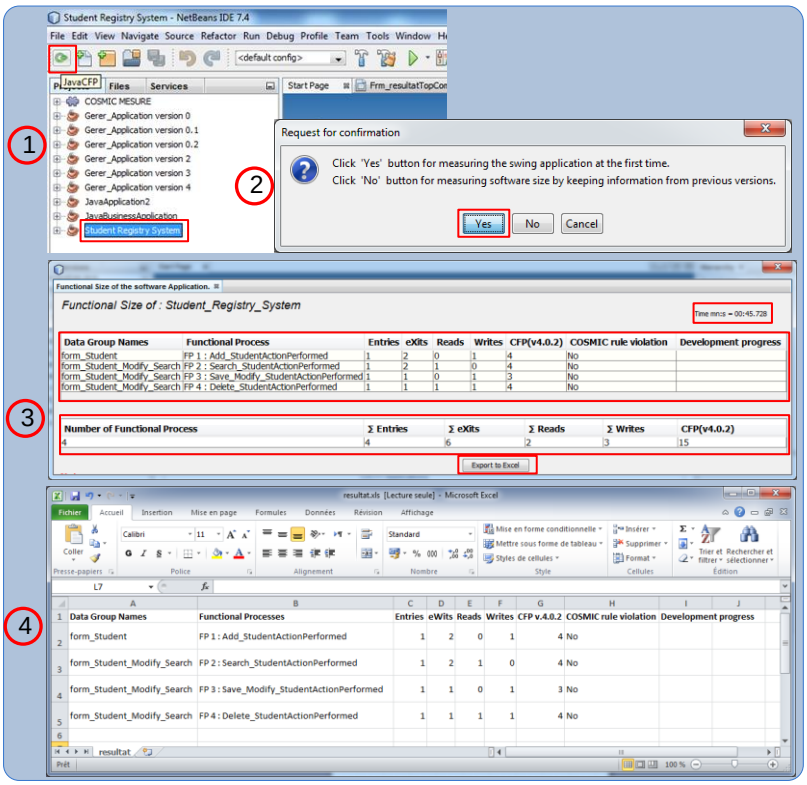

Fig. 4. Snapshot of "JavaCFP" Tool with Case Study "C-REG"

Interface 1 is used to select the piece of code to be measured. Of course, the code should be implemented in NetBeans IDE. By clicking the icon "JavaCFP" plugin in the toolbar, a dialog box appears in interface 2 asking whether the tool user wishes to keep track or not of the previous measurement results (e.g., the software functional size can be generated while the coding is in progress and after the coding is completed). In interface 3, "JavaCFP" automatically generates the functional size of the piece of code in terms of CFP units at a point in time during its development. Interface 3 also provides detailed information about the functional size of each data movement type (E, X, R, and W). Note that the real execution time (processing time) per second is provided as the program is executed. For instance, interface 3 exhibits the processing time which is about 46 seconds for 15 CFP developed including 997 LOC. These measurement results can be documented into Microsoft Excel format through Interface 4.

5.2 Comparison

In this section, we use some of the concepts from the ISO International Vocabulary on Metrology (e.g., measurement method, measurement procedure, devices for measurement, accuracy of measuring devices, and indicating measuring instrument) [23] [24] [25] as criteria for comparison among the proposed COSMIC-based tools.

- Measurement method: since the COSMIC method behind the tool is well-designed, all the proposed tools [13] [14] [16] should provide the adequate functional size of swing applications.
- Measurement procedure: it is necessary to run GUI for all the proposals [13] [16] [14].
  - In [13], the measurement procedure is divided into three steps. The first step consists of recording the data movement. The second one includes the discovery of functional processes. The third step consists of counting the data movements in each functional process to provide the code functional size.
  - In [16] the authors start with creating notifications for COSMIC data movement sub-types. After that, they identify the functional processes and calculate their CFP sizes.
  - In [14] four steps are proposed: (1) generate sequence diagrams from code (Java source or binary), (2) get its text version to (3) capture data movements and data manipulation through AspectJ, (4) extract the functional execution traces from user execution.
  - Our JavaCFP tool identifies the FUR of the selected Java Swing application (without any execution) directly from the source code in the "Source Packages". Then, for each .java file (JFrame form), we identify the triggering event from Event listener attached to "Swing Controls". Then, we identify the functional process (FP) from invoked callback_methods. For each FP, we identify the different data attributes belonging to the data groups where each data group describes the same object class. Finally, the details of measuring the functional size of each FP are provided by using measurement formulae (1) and (2).
- Devices for measurement
  - [13] proposed a semi-automated measurement library for GUI Java application.
  - [16] proposed an automated tool "static code installer" that used a library component.
  - [14] proposed an automated instrument named "Cosmic Solver".
  - we proposed an automated JavaCFP tool that can be used directly either when the code is being developed or after code completion.
- Measurement accuracy: a measurement is said to be more accurate when it has a smaller measurement error.
  - In [13], the authors reported that the use of the library from a "student registration" system led to a corresponding ratio of 92% between the code functional sizes calculated automatically against those calculated manually.

- In [16] the automatic measurement converges by 94% compared to the manual one.
- In [14] numbers are quite similar with 96,8% convergence.
- In our proposal, the measurement results obtained manually and automatically by "JavaCFP" are similar: it gives the details with 100% of measurement accuracy, including the same functional processes with their detailed descriptions. It guarantees the accuracy of measurement at any time during the code implementation..

– Indicating measuring instrument: compared to the other proposed tools in [13] [14] [16], our JavaCFP tool provides an indication of deviations (e.g., programming defects) when there is a "COSMIC rules violation" illustrating the part of the code not yet completed, and the "development progress" that indicates the current development of a new or modified functional process.

6 Conclusions

This paper has presented the JavaCFP tool that can be used as a basis to assist the developers in writing their java swing applications and generating the COSMIC functional size of that application at any time during its development lifecycle. This JavaCFP tool can be used by developers having different programming styles and even when they are not familiar with COSMIC method. A condition is that each event listener interface should be associated to only one object.

Throughout this paper, we first extracted the java concepts that can be mapped with COSMIC concepts, and generated the mapping rules, measurement rules and formulae that allow measuring manually the COSMIC functional size of swing application. After that from these formulae, we proposed the "JavaCFP" tool to help developers generate automatically the COSMIC size of swing application while it is being written. The benefits of using "JavaCFP" are not limited to generating automatically the software functional size: JavaCFP also detects which functionalities are omitted or added within a method (e.g. missing Entry), and identifies the current development progress (new or modified functional process).

Tool users can also verify the completeness of the implemented functionality against specified FUR for tracking project progress purposes. The COSMIC size of java swing application can be measured very precisely when the FUR are described without uncertainty.

In further works, we will use machine learning concepts for the identification of the objects of interest in the swing code. We will investigate the variation of measurement results when functional changes and improvements occurred in the code. We will also investigate the benefits of the proposed tool in industry, including for measuring the project productivity and the data collection for future estimation needs.

References

1. A. Abran, Software Project Estimation: The Fundamentals for Providing High Quality Information to Decision Makers, Wiley-IEEE Press, 2015, p. 261.
2. R. E. Fairley, Managing and Leading Software Projects, Wiley-IEEE Computer Society (c) 2009, ISBN 978-0470-29455-0, p510, February 2009.
3. M. S. Jenner, "Automation of Counting of Functional Size Using COSMIC-FFP in UML," *12th International Workshop Software Measurement- IWSM*, Magdeburg, Germany, 2002.
4. Z. Li, M. Nonaka, A. Kakurai and M. Azuma, "Measuring functional size of interactive software: a support system based on XForms-format user interface specifications," *Third International Conference (QSIC'03)*, Dallas, Texas, 2003.
5. N. Condori-Fernandèz, S. Abrahão and O. Pastor, "On the Estimation of Software Functional Size from Requirements Specifications," *Journal of Computer science and Technology. Vol. 22, pp. 358-370. URL:http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.144.5364&rep=rep1&type=pdf*, 1 Mai 2007.
6. S. Barkallah, A. Gherbi and A. Abran, "COSMIC Functional Size Measurement Using UML Models," in *Software Engineering, Business Continuity, and Education, Communications in Computer and Information Science, International Conferences ASEA, DRBC and EL 2011, Held as Part of the Future Generation Information Technology Conference, FGIT 2011, Volume 257*, 2011.
7. H. Soubra, A. Abran, S. Stern and A. Ramdan-Cherif, "Design of a Functional Size Measurement Procedure for Real-Time Embedded Software Requirements Expressed using the Simulink Model," *Joint Conference of the 21st Int'l Workshop on Software Measurement and 6th Int'l Conference on Software Process and Product Measurement - IWSM-MENSURA*, Nara, Japan, 2011.
8. H. Soubra, A. Abran and A. Ramdane-Cherif, "Verifying the Accuracy of Automation Tools for the Measurement of Software with COSMIC -- ISO 19761 Including an AUTOSAR-Based Example and a Case Study," *Joint Conference of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement*, DOI: 10.1109/IWSM.Mensura.2014.26, Rotterdam, The Netherlands, 6-8 Oct. 2014.
9. V. Bévo, G. Lévesque and A. Abran, "Application de la méthode FFP à partir d'une spécification selon la notation UML: compte rendu des premiers essais d'application et questions," *9th International Workshop Software Measurement - IWSM, Lac Supérieur*, Canada, 1999.
10. S. Azzouz and A. Abran, "A proposed measurement role in the Rational Unified Process (RUP) and its implementation with ISO 19761: COSMIC-FFP," *Software Measurement European Forum - SMEF*, Rome, Italy, 2004.
11. G. Grau and X. Franch, "Using the PRiM method to Evaluate Requirements Models with COSMIC-FFP," *International Conference on Software Process and Product : Measurement -IWSM-Mensura*, Mallorca, Spain, 2007.
12. B. Marín, O. Pastor and G. Giachetti, "Automating the Measurement of Functional Size of Conceptual Models in an MDA Environment," *Functional Size of Conceptual Models in*

an MDA Environment. the international conference on Product-Focused Software Process Improvement (PROFES '08), Andreas Jedlitschka and Outi Salo (Eds.). Springer-Verlag, Berlin, Heidelberg, 2008.

13. A. Akca and A. Tarhan, "Run-time Measurement of COSMIC Functional Size for Java Business Applications: Initial Results," *International workshop on software Measurement and 7th International Conference on Software Process and Product Measurement - IWSM-MENSURA*, Assisi, Italy, 17-19 Oct. 2012.
14. M. A. Sağ and A. Tarhan, "Measuring COSMIC Software Size from Functional Execution Traces of Java Business Applications," *Joint Conference of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement - IWSM-MENSURA*, October 2014, pp. 272-281.
15. M. A. SAĞ and A. TARHAN, "COSMIC Solver: A Tool for Functional Sizing of Java Business Applications," *Balkan Journal of Electrical and Computer Engineering*, vol. 6, pp. 1-8, 2018.
16. R. Gonultas and A. Tarhan, "Run-Time Calculation of COSMIC Functional Size via Automatic Installment of Measurement Code into Java Business Applications," *41st Euromicro Conference on Software Engineering and Advanced Applications*, 2015.
17. S. Bagriyanik and A. Karahoka, "Automated COSMIC Function Point measurement using a requirements engineering ontology," *Information and Software Technology*, 2016.
18. A. Tarhan, B. Özkan and G. C. İçöz, "A Proposal on Requirements for COSMIC FSM Automation from Source Code," *Joint Conference of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement -IWSM-MENSURA*, 5-7 Oct 2016, Berlin. pp. 195–200.
19. ISO/IEC 14143-1, *Information technology - Software measurement - Functional size measurement - Part 1: Definition of concepts*, International Organization for Standardization, Geneva, 2007.
20. COSMIC, The COSMIC Functtionall Size Measurementt Method: Measurementt Manuall Version 4.0.2, C. Symons and A. Lesterhuis, Eds., The COSMIC group. URL: www.cosmic-sizing.org, December 2017.
21. A. Akca and A. Tarhan, "Run-Time Measurement of COSMIC Functional Size for Java Business Applications: Is It Worth the Cost?," *Joint Conference of the 23rd International Workshop on Software Measurement and Eighth International Conference on Software Process and Product Measurement - IWSM-MENSURA*, Ankara, Turkey, 2013.
22. A. Lesterhuis, A. Abran and C. Symons, "Course Registration ('C-REG') System Case Study, Version 2.0.," https://cosmic-sizing.org/publications/course-registration-c-reg-system-case-study/, December 2015.
23. A. Abran, Software Metrics and Software Metrology, J. W. &. Sons, Ed., Wiley-IEEE Computer Society Press, 2010.
24. A. Sellami and A. Abran, "Measurement and metrology requirements for empirical studies in software engineering," *10th International Workshop on Software Technology and Engineering Practice, pp 185-192, doi={10.1109/STEP.2002.1267631}*, 2002.
25. A. Sellami, "Processus de vérification des mesures de logiciels selon la perspective de métrologie". Doctoral thesis, École de technologie supérieure, Montréal, Canada 2005.