STATIC CODE ANALYSIS FOR SOFTWARE QUALITY IMPROVEMENT: A CASE STUDY IN BCI FRAMEWORK DEVELOPMENT

Indar Sugiarto
Department of Electrical Engineering – Petra Christian University
Jl. Siwalankerto 121-131, Surabaya – 60236
Email: indi@petra.ac.id

ABSTRACT: This paper shows how the systematic approach in software testing using static code analysis method can be used for improving the software quality of a BCI framework. The method is best performed during the development phase of framework programs. In the proposed approach, we evaluate several software metrics which are based on the principles of object oriented design. Since such method is depending on the underlying programming language, we describe the method in term of C++ language programming whereas the Qt platform is also currently being used. One of the most important metric is so called software complexity. Applying the software complexity calculation using both McCabe and Halstead method for the BCI framework which consists of two important types of BCI, those are SSVEP and P300, we found that there are two classes in the framework which have very complex and prone to violation of cohesion principle in OOP. The other metrics are fit the criteria of the proposed framework aspects, such as: MPC is less than 20; average complexity is around value of 5; and the maximum depth is below 10 blocks. Such variables are considered very important when further developing the BCI framework in the future.

Keyword: static code analysis, software quality, BCI framework.

INTRODUCTION

A brain computer interface is a system that includes a means for measuring neural signals from the brain, a method/algorithm for decoding these signals and a methodology for mapping this decoding to a behavior or action. In an applicable EEG-based BCI system, there are at least seven major components which are required to be synchronized: data acquisition, signals database/storage, feature processing (extraction and classification), visualization (temporal or spatial), command generation for actuator(s), command database, and feedback acquisition. The following diagram shows an example of high level data abstraction used in a BCI Framework [1].

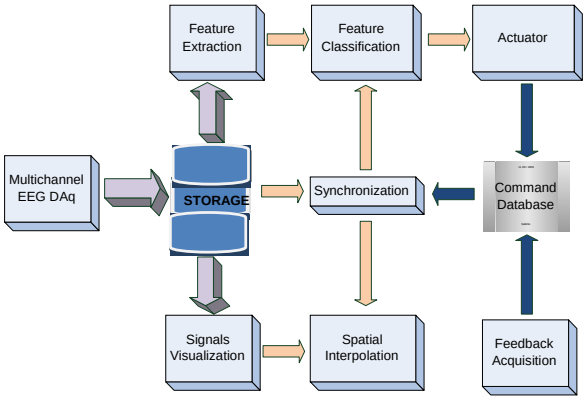

Figure 1. High level data abstraction of a BCI framework.

There are several important parameters necessary for developing a BCI framework which supports future application of the framework which comply the standard quality for software development defined by ISO 9126 (which is being superseded by ISO 25000:2005), such as: functionality, reliability, usability, efficiency, maintainability, and portability. These attributes, which are part of software quality assurance standards used in the software industry, should be considered important when developing the BCI framework. This research is designed to fulfill those general requirements although not in specific term since those attributes are abstract and only have metrics for completed framework [3]. There is, however, a systematic approach to assess quality of the software in this research which is classified as software quality control. This systematic approach, which is called software testing, is an empirical technical investigation conducted to provide information about the quality of the software under test based on several software metrics. Whereas software quality control is a control of products, software quality assurance is a control of processes. In this research, software testing is performed in static code analysis.

Apart of aforementioned attributes, we propose the following aspects when developing the BCI framework:

- Real-time, compilation-based and object oriented programming language. In this case, C++ is chosen since it is well-known for its speed, portability and abstraction.

- Logical separation of modules. The inter-operable components of the system should work (almost) independently in its own thread or process (including remote system).
- Open architecture: software components should be open for extensions but closed for changes. Extensions should be possible without any changes of the current source code.
- Support platform independent paradigm. Although we cannot create fully platform independent system due to driver support problem at the moment, the program should directed to support platform independent in the future development of BCI framework.
- Well defined documentation which supports abstraction level of modular structure and test-driven design.
- Easy to use and well-defined interface.

There are several key points which are very important to implement a real-time framework using standard PC based on the above approaches. Although every Operating System may different in realizing or supporting these concepts, this difference may not look so big if the underlying programming language is the same. In this paper, we use C++ as the programming language to build the framework which is based on general assumption that the well-known C++ is fast enough and close enough to the machine language paradigm. The Qt framework, which provides fast and robust GUI visualization of programs, is used in this research. Thus, the static code analysis in this paper will be described based on C++ programming paradigm.

This paper is organized as follows. After giving a brief explanation about the motivation background, the paper continues with the description of the method and followed by the implementation result. The discussion about the result will be closed with conclusions.

METHOD

There are several methods that can be used to test the BCI framework software in order to maintain such level of software quality. At some points, software testing can be considered as part of Software Quality Control. After finishing the design process (also completing verification procedure), the program should undergo validation process through several testing procedures, statically or dynamically. Static testing is a form of software testing where the software's code is inspected without actually running the software. Dynamic testing, on the other hand, will execute the software and inspect the behavior of the software during run-time. In this paper, static code analysis is performed by measuring several metrics in order to collect information about software complexity and vulnerability.

There are several important software metrics which are measured in this paper. Software metric is a quantitative parameter that is commonly used in software quality assessment. It is common in software engineering that one method may differ from another due to different programming language scheme and structure [4], [5]. For example, the metrics in C++ source files can be ambiguous when conditional compilation is used to define different versions of a substructure. Therefore, some software tools ignores all conditional #else clauses and counts metrics only in the code that lies between the #if... and #else preprocessor directives. Not all of metrics for C++ will be used in this paper; only the following metrics are used (Table 1).

Especially for measuring software complexity, a significant complexity measure increase during testing may be the sign of a brittle or high-risk module. In this paper both approaches, McCabe and Halstead, are used. Halstead measures have been criticized for a variety of reasons, among them the claim that they are a weak measure because they measure lexical and/or textual complexity rather than the structural or logic flow complexity exemplified by Cyclomatic Complexity measures. However, they have been shown to be a very strong component of the Maintainability Index measurement of maintainability [7]. In particular, the complexity of code with a high ratio of calculational logic to branch logic may be more accurately assessed by Halstead measures than by Cyclomatic Complexity, which measures structural complexity.

There are three software tools which are used in this paper: SourceMonitor, LocMetrics, and Crystal REVS. SourceMonitor and LocMetrics are freeware and available for free download, while Crystal REVS is proprietary software but it offers an evaluation copy. All of that software can be downloaded from the following website:

- http://www.campwoodsw.com/sourcemonitor.html
- http://www.locmetrics.com
- http://www.sgvsarc.com/

To maintain the quality of the program during development, the following rules which are based on our proposed approach mentioned in the introduction are used:

- Method per class must be less than 20 to avoid superclass structure. Preferably, the maximum MPC (methods per class) is kept below scale of 10 in order to maintain low coupling in the module.
- Average complexity is kept as low as possible, preferable below scale of 5.
- The maximum depth is kept below 10 blocks to reduce function overhead.

Table 1. C++ metrics to be used for static code analysis.

Metrics	Description
Lines of Code (LOC)	This metric counts the lines of non-blank and non-comment source code. LOC was the earliest metrics to be used in computer science and still be used for many decades even until now, although some researchers thought that its meaning is absurd. In C++, computational statements are terminated with a semicolon character. Branches such as if, for, while and goto are also counted as statements. The exception control statements try and catch are also counted as statements. Preprocessor directives #include, #define, and #undef are counted as statements. All other preprocessor directives are ignored. In addition all statements between each #else or #elif statement and its closing #endif statement are ignored, to eliminate fractured block structures.
Percent Lines with Comments	The lines that contain comments, either C style (/*...*/) or C++ style (//...) are counted and compared to the total number of lines in the file to compute this metric.
Methods per Class	Both inline and non-inline class and template class implementations are counted. This metric is an overall average for all class and template method implementations in a file or checkpoint, computed as the total number of methods divided by the total number of classes and templates for which method implementations are found.
Average Statements per Method	The total number of statements found inside of methods found in a file or checkpoint divided by the number of methods found in the file or checkpoint.
Software Complexity	The complexity value of the most complex method or function in a file. The most common method to measure complexity is by using cyclomatic complexity of the directed acyclic graph which represents the flow of control within each function. First proposed by McCabe [6] as a measure of the minimum number of test cases to ensure all parts of each function are exercised. It is now widely accepted as a measure for the detection of code which is likely to be error-prone and/or difficult to maintain. Another complexity value that can be used for software assessment is Halstead's Complexity.
Average Complexity	The Average Complexity metric is a measure of the overall complexity measured for each method (and, if present, each function) in a file or checkpoint. It is computed as a simple arithmetic average of all complexity values measured for a file or project.
Block Depth	Block Depth is a measure of how many nested blocks are exists within one upper level block. Maximum Block Depth is calculated from the top level of the source code, but namespaces are not included in this block depth metrics. Average Block Depth is the average nested block depth weighted by depth.

EXPERIMENT RESULT

In our BCI framework, we develop the CssvepLed class which is responsible for handling only the SSVEP part of the BCI framework and has been started since January 2008 [1], [8]. It was then integrated into CBCIFrameWork, which is the final framework of the BCI system [2], in the beginning of April 2008. A new Cp300 class, which is responsible for handling the P300 part of the BCI framework, was integrated into the CBCIFrameWork also in April 2008 together with the letter matrix for the spelling application.

One of the applications of our BCI framework is the Spelling Program. The first version the program, which is called The Speller, was created on November 2007 and it uses the aforementioned CssvepLed class [9]. After the development of the CBCIFrameWork, The Speller was renamed as Speller_v2.xx (where xx is the minority updates to the framework). Speller_v2.xx is an integrated program which accommodates both SSVEP and P300 type of BCI. Until now, this integrated system has reached its version 2.7. It means that there are seven updates / changes introduced to the framework since its starting. The following table gives summary from all seven sub-version of the integrated system developed in this research. The values in this table are generated by SourceMonitor version 2.4.

Table 2. Summary of software metrics generated for seven sub-version of the program

	NOM	LOC	%COM	MPC	Average LOC/Method	Average Complexity	Max Depth	Average Depth
Speller_v2.0	9	1790	9.80	6.75	15.70	5.42	7	1.93
Speller_v2.1	10	2155	11.90	6.44	18.20	6.05	7	2.20
Speller_v2.2	10	2329	9.00	7.00	20.00	6.48	8	2.38
Speller_v2.3	16	3492	7.40	7.07	17.30	5.22	8	2.12
Speller_v2.4	18	4322	9.30	7.06	18.60	5.17	8	2.05
Speller_v2.5	18	4418	9.20	7.06	19.00	5.19	8	2.04
Speller_v2.6	18	5360	8.70	8.29	19.90	5.27	8	1.98

Note: NOM = Number of Modules, LOC = Lines of Code, %COM = Percent Lines with Comment, MPC = Methods per Class

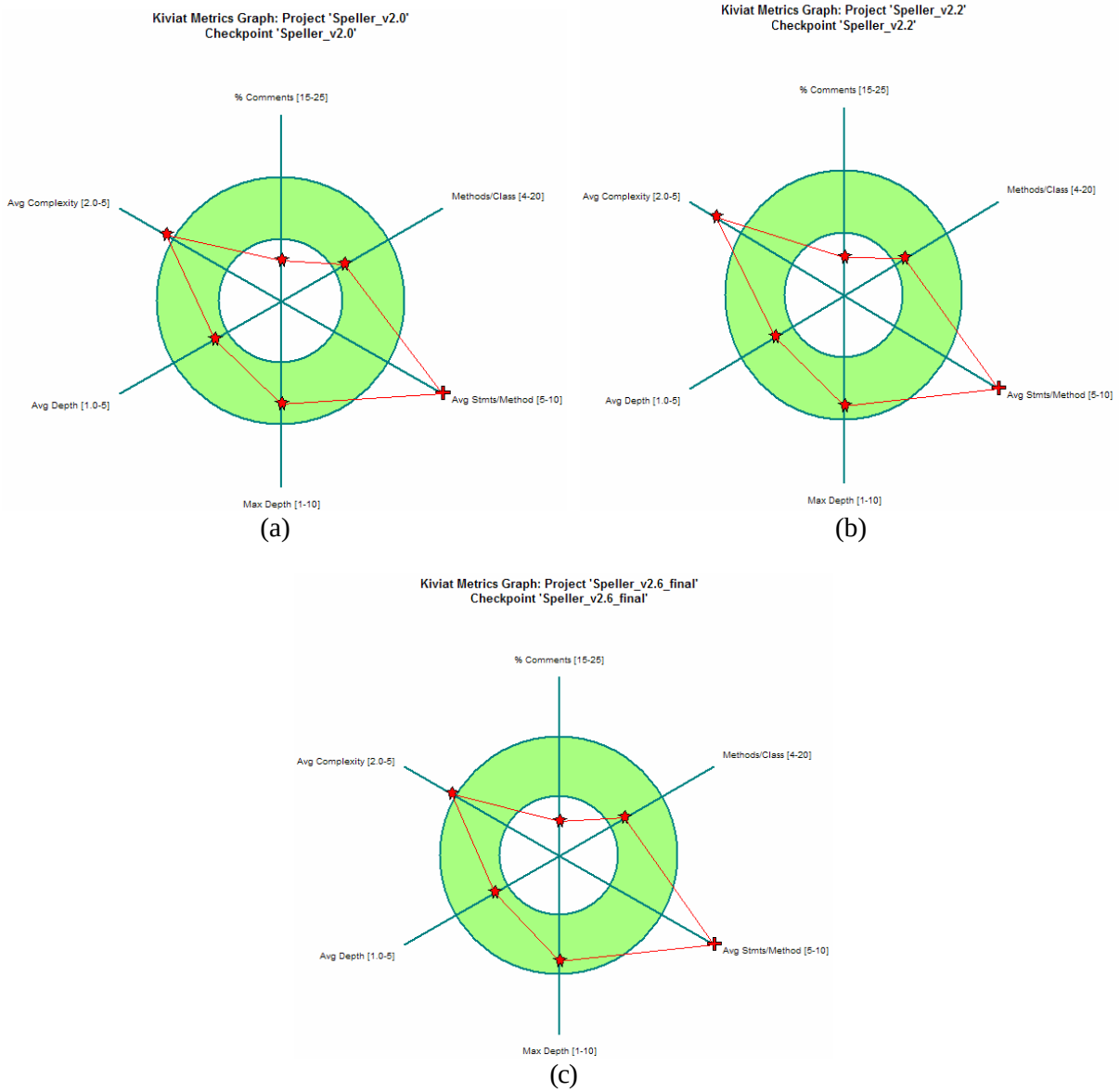

Figure 2. Kiviat Diagram of three version of the integrated system: a) Speller_v2.0, b) Speller_v2.2, c) Speller_v2.7

The following picture shows Kiviat Diagram for Speller_v2.0, Speller_v2.2, and Speller_v2.7. These Kiviat Diagrams are generated according to data presented in Table 2 using SourceMonitor program.

From Table 2 and Figure 2, it can be concluded that:

- The process development of the program is in a good direction, since the average complexity of overall program tends to lower. However, it is very difficult to maintain the average complexity below scale 5 due to individual complexity in each class. Also, Methods per Class and Average Depth are kept in range, although they tend to increase.
- The Number of Methods per Class is low, thus it will increase the number of LOC per method. This is undesirable because it violates Single-Responsibility principle of Object Oriented Design and should be optimized in the future.
- The percent of Lines with Comments is very low. Unfortunately, some modules in the program are generated automatically by the Qt's compiler and we cannot add specific comments to those modules. Low percentage in this value means that the program, in general, lacks of self-explanation. It means that, it becomes difficult to understand or modify the program by other person. However, increasing the number of comments in another module beside of those auto-generated modules by Qt will yields in a source code full of comments and somehow makes it less clear to understand the main program structure.

To make final conclusion from static code analysis for the final version of the program in this paper, the Cyclomatic Complexity from McCabe and Difficulty Metric from Halstead are also calculated. The following tables are result analysis from program LocMetrics and Crystal REVS which are applied to Speller_v2.7.

Table 3. McCabe's Complexity measurement result for Speller_v2.7.

Modules	McCabe's Complexity					
	Cyclomatic Complexity v(G)	Design Complexity iv(G)	Essential Complexity vd(G)	Cyclomatic Density id(G)	Design Density id(G)	Essential Density ed(G)
BCIGlobal	0	0	0	0	0	0
CConfig	31	4	25	0.15	0.13	0.81
CGraphicsFrameWork	41	3	15	0.14	0.07	0.37
CImpedance	3	1	3	0.05	0.33	1
CMainDialog	49	14	19	0.15	0.29	0.39
Cp300	113	18	34	0.19	0.16	0.3
Cp300Thread	20	8	13	0.17	0.4	0.65
CParserHelper	24	2	26	0.36	0.08	1.08
CProtocol	1	1	2	0.05	1	2
CssvepLed	120	29	39	0.16	0.24	0.32
CssvepThread	4	1	5	0.11	0.25	1.25
CssvepTimer	11	6	5	0.22	0.55	0.45
CSystemLogging	25	7	11	0.14	0.28	0.44
CTcpIp	9	4	8	0.11	0.44	0.89
main.cpp	1	1	1	0.08	1	1
ui_Config	1	1	1	0.05	0.07	0.95
ui_Impedance	1	1	1	0.03	0.05	0.5
ui_Protocol	1	1	1	0.06	0.07	0.95

Table 4. Halstead's Complexity measurement result for Speller_v2.7.

Modules	Halstead's Complexity				
	Length (N)	Vocabulary (n)	Volume (v)	Difficulty (d)	Effort (e)
BCIGlobal	6	6	10.75	0	0
CConfig	834	137	4103.26	4	17518.81
CGraphicsFrameWork	927	194	4888.3	13	66094.6
CImpedance	170	41	631.31	1	932.28
CMainDialog	966	211	5169.89	11	58226.59
Cp300	2546	389	15183.27	14	225398.7
Cp300Thread	231	59	941.91	11	11124.89
CParserHelper	172	46	658.53	2	1915.71
CProtocol	4	4	5.5	0	0
CssvepLed	3256	440	19818.54	14	279808.4
CssvepThread	9	4	12.48	1	12.48
CssvepTimer	49	23	153.64	7	1109.08
CSystemLogging	397	97	1816.16	10	19029.7
CTcpIp	101	49	393.07	12	5089.27
main.cpp	26	18	75.15	3	250.5
ui_Config	2258	369	13346.58	11	158342.1
ui_Impedance	659	121	3160	6	19290.75
ui_Protocol	2082	348	12184.29	9	109852

Combining both Table 3 and Table 4 will result in the following diagram.

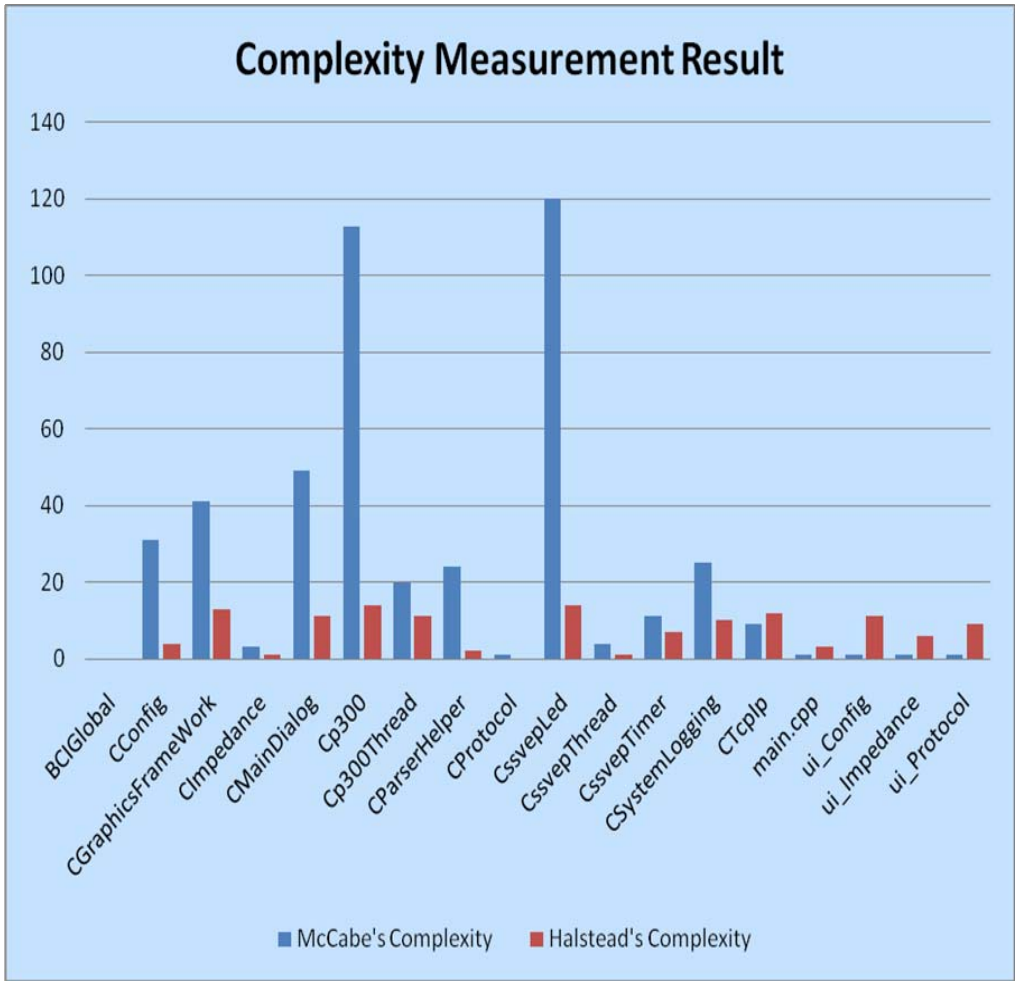

Figure 3. Complexity Measurement Result for both McCabe and Halstead Metrics.

DISCUSSION

It can be seen from Figure 3 that CssvepLed and Cp300 are two most complex members of CBCIFrameWork. The module with high complexity value tends to be difficult to maintain and also has high defect density. This information is very important during development phase, especially if the framework will be further developed in a team work. Using such kind of graph, every member of the team will know which module or part of the program requires more attention and effort. We will also know that we are not supposed to create an interface between complex modules directly (e.g. between Cp300 and CssvepLed) because it will create a new overhead and also violates cohesion principle in OOP (Object Oriented Programming). One way to reduce complexity of CssvepLed and Cp300 is by re-structuring those classes into smaller classes and then use inheritance and encapsulation principle of OOP to increase their cohesion. This approach should be explored further if CBCIFrameWork will be employed for future development.

It also can be seen that modules such as ui_Config, ui_Impedance, and ui_Protocol, which are produced using Qt's platform, have intrinsic characteristic in that they have low cyclomatic complexity but high difficulty metric. That is why Kiviat Diagram in Figure 2 shows very low value for the percentage of Lines with Comments. For person who doesn't have any experience with Qt, reading those modules will be very difficult.

CONCLUSION

A method for assuring the quality of the software through systematic approach using static code analysis has been presented. The method has been applied during the development phase of a BCI framework. In the proposed method, we evaluate several software metrics which are based on the principles of object oriented design. Since such method is depending on the underlying programming language, we describe the method in term of C++ language programming whereas the Qt platform is also currently being used. One of the most important metric is so called software complexity. Applying the software complexity calculation using both McCabe and Halstead method for the BCI framework which consists of two important types of BCI, that is SSVEP and P300, we found that there are two classes in the framework which have very complex and prone to violation of cohesion principle in OOP. The other metrics are fit the criteria of the proposed framework aspects, such as: MPC is less than 20; average complexity is around value of 5; and the maximum depth is below 10 blocks. Such variables are considered very important when further developing the BCI framework in the future.

ACKNOWLEDGEMENT

The authors would like to thank Dr. Brendan Allison and Prof. Axel Gräser from Institute for Automation at University of Bremen for their advice, and all volunteers for taking part in the experiment.

REFERENCES

1. Sugiarto, I. and Putro, I. H. 2009. *Application of Distributed System in Neuroscience, A Case Study of BCI Framework*. The 1st International Seminar on Science and Technology 2009 (ISSTEC2009). Yogyakarta: Universitas Islam Indonesia.
2. Sugiarto, I. 2009. *Display and Feedback Approach for BCI System*. Master Thesis. University of Bremen.
3. Jalote, P. 2005. *An Integrated Approach to Software Engineering*. Springer.
4. Borsoi, B. T. and Jorge L.B. 2008. *A Method to Define an Object Oriented Software Process Architecture*. 19th Australian Conference on Software Engineering. IEEE.
5. Taylor, R.N., Medvidovic, N., Anderson, K.N. and Whitehead Jr., E.J. 1996. A Component and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*. Vol. 22(6): 390-406.
6. McCabe, T. 1976. A Complexity Measure. *IEEE Transactions On Software Engineering*. Vol. 2(4): 308-320.
7. VanDoren, E. 1997. *Halstead Complexity Measure*. Last accessed May 8, 2008 from http://www.sei.cmu.edu/str/descriptions/halstead.html.
8. Valbuena, D., Sugiarto, I and Gräser, A. 2008. *Spelling with the Bremen Brain-Computer Interface and the Integrated SSVEP Stimulator*. in proc 4th International Brain-Computer Interface Workshop and Training Course 2008. Austria: Verlag der Technischen Universität Graz. pp. 291-296.
9. Sugiarto, I., Allison, B. Z. and Gräser, A. 2009. *Optimization Strategy for SSVEP-Based BCI in Spelling Program Application*. International Conference on Computer Engineering and Technology 2009 (ICCET 2009). Singapore: International Association of Computer Science and Information Technology.