**Available Online at www.ijcsmc.com**

**International Journal of Computer Science and Mobile Computing**

**A Monthly Journal of Computer Science and Information Technology**

**ISSN 2320–088X**

*IJCSMC, Vol. 3, Issue. 11, November 2014, pg.89 – 101*

**RESEARCH ARTICLE**

# Cyclomatic Complexity Metrics for Software Architecture Maintenance Risk Assessment

**RONALD TOMBE ***
**Dr. GEORGE OKEYO ****
**Dr. STEPHEN KIMANI *****
*Dept. of Computing Sciences, Kisii University, Kenya
**Dept. of Computing, Jomo Kenyatta University of Agriculture and Technology, Kenya
***Dept. of Computing, Jomo Kenyatta University of Agriculture and Technology. Kenya

***Abstract: A well-defined strategy is a key to successful software project maintenance as it enables change management and minimize risks associated with the future operation of the software. Software maintainers usually are not engaged in the initial software development cycle. Before maintainers can modify a program, they must understand how it operates. This research established that there exist various research gaps in literature in the architecture analysis methods and of interest was that of maintainability risk assessment at the architecture level and as a result a method was proposed to address the maintainability risk assessment research gap identified. An experiment was designed to validate the developed method for maintainability risk assessment at the architecture level, from the results of the experiment it was established modules participating in design patterns are less change prone, they promote easy of change; hence classes participating in patterns should require fewer changes and that McCabe Cyclomatic Complexity measure is useful in determining the complexity of the implementation mechanism at the architecture level which is useful for the maintainability risk assessment at the architecture level.***

***Key Words: Software Architecture, design patterns, Cyclomatic complexity metrics, Risk assessment***

## 1.0 Introduction

Garlan (2000) defines software architecture as the structure of components, and their interrelationships, and the principles and guides that control the design and evolution in time. According to (Bass, 2003) software architecture is as an abstract structural description of the software system in terms of its main components and the relationships among them. Studies on quantitative assessment of software architectures are gaining importance due to their role in assessing the quality of architecture enhancements (Sant'Anna, 2007). IEEE 1471 standard defines software architecture as the fundamental organization of a system embodied in its components, their relationships to each

other and to the environment and the principles guiding its design and evolution (Emery, 2008). From this definition, the component and the connector are reinforced as the central concepts of software architecture. A component can be as simple as an object, a class, or a procedure, and as elaborate as a package of classes or procedures. Connectors can be as simple as procedure calls or as elaborate as client-server protocols, links between distributed databases, or middleware.

Software maintenance is classified into adaptive, corrective, preventive and perfective (Somerville, 2008). Most organizations are concerned about the costs of software maintenance, for it has been increasing steadily and many companies spend approximately 65% of their software budget on maintenance (Abdelmoez, 2006). The process of risk assessment is useful in identifying complex modules that require detailed inspection, estimating potentially troublesome modules. According to the NASA-STD-8719.13A, risk is a function of the anticipated frequency of occurrence of an undesired event, the potential severity of resulting consequences, and the uncertainties associated with the frequency and severity (Abdelmoez, 2006).

Emphasis on software architecture is being put on design patterns (Shaik, 2005), thus the traditional practice of ad-hoc software construction is slowly shifting towards pattern- oriented development.

The architecture allows for multiple independent and loosely-coupled component implementation mechanisms. Risk assessment and analysis for software architectures is motivated by the fact that different operators and vendors may choose to deploy/develop different mechanisms to achieve the same end, and there need to be different mechanisms to solve problems at different places in a networked environment. Furthermore, implementation bugs or configuration errors could potentially render an implementation ineffective.

### 1.1 Study Hypothesis

The experiment was designed in such a way to examine the relationship between single module properties and change proneness. The experiment tested the following hypothesis:

***H1***: Larger modules are more change prone. A larger module has more functionality, thus there is greater likelihood that some functionality in the module will need to be corrected or enhanced.

***H2***: Modules participating in design patterns are less change prone. Patterns are designed so that changes are made via subclasses or by adding new participant classes rather than modifying already present classes. Patterns promote easy of change; hence classes participating in patterns should require fewer changes.

## 2.0 Literature Review

### 2.1 Architecture-Level Prediction of Software Maintenance (ALPSM)

The goal of ALPSM is to predict the maintenance effort required to address a change scenario (Bengtsson, 1999).The main contribution of this method consist of the architecture level where this prediction is performed. ALPSM defines a maintenance profile, like a set of change scenario tasks. A scenario describes an action, or sequence of actions that might occur as related to the system. Hence a change of scenario describes a certain maintenance tasks. Using the maintenance profile, the architecture is evaluated using the scenario describes a certain maintenance effort for a software system can be estimated. The method has a number of inputs the requirements specifications, the design of the architecture, expertise from software engineers and possibly historical maintenance data. This method analyses maintainability by looking at the impact of scenarios. It uses the size of changes as a predictor for the effort needed to adopt the system to a scenario. The ALPSM does not to address risk assessment (Tombe et al, 2014) thus the need to improve the method so as to incorporate the risk assessment aspect during software maintenances.

### 2.2 Method for Software Maintenance Risk Assessment at the Architecture Level (MSMRAAL)

The Architecture level prediction software maintenance method (ALPSM) discussed ***in section above*** does not provide mechanisms to address the risks that are associated with the maintenance changes (Tombe et al, 2014).

The method for software maintenance risk assessment at the architecture level consists of the following steps.

1. *Identify categories of maintenance tasks from the scenarios. Model the scenarios using UML specifications:*
2. *Synthesize scenarios:* For each of the maintenance tasks, a representative set of scenarios will be defined.
3. *Map the scenarios into the architectural design: F*or each scenario determines the components that are affected and to what extent they will be changed, this results in the size of the impact of the realization of the scenario.
4. Map the Participating classes of the scenarios as presented in UML specifications model(s) to a published design pattern that best matches the Model.
5. *Risk assessment:* Establish that the impact of a change scenario; estimate the risk on the ripple effects of the changes (maintenance) to be made on a component in respect with the interacting components in order to predict the overall risk that might be associated to during the maintenance of a system

In this research, we use McCabe's metrics for maintainability risk assessment on the implementation mechanisms used in the experiment setup for this research. This is discussed in the **section 2.3**. The reason as to why the McCabe's metrics is used is because it will give a quantitative measure of the results on the risk assessments of the software component implementation mechanism used at the architecture level.

### 2.3 McCabe's Cyclomatic Complexity

McCabe (1976), views program complexity related to the number of control paths through a program module. McCabe derived a software complexity measure from graph theory using the definition of the cyclomatic number which corresponds to the number of linearly independent paths in a program. It is intended to be independent of language and language format. This measure provides a single number that can be compared to the complexity of other programs.

McCabe's cyclomatic complexity is an indication of a program module's control-flow complexity and has been found to be a reliable indicator of complexity in large software projects (Ward, 1989). Considering the number of control paths through the program, a 10-line program with 10 assignment statements is easier to understand than a 10-line program with 10 if-then statements. *MCC* is defined for each module to be M=E − *N* + X, where M is the McCabe Cyclomatic Complexity (MCC) metric, E is the number of edges, N is the number of nodes or decision points (conditional statements), and X is the number of exits (return statements) in the graph of the function respectively. Control flow graphs (CFC) describe the logic structure of software modules. The nodes represent computational statements or expressions, and the edges represent transfer of control between nodes. Each possible execution path of a software module has a corresponding path from the entry to the exit node of the module's control flow graph. The advantages of the CFC metric is that it can be used as a maintenance and quality metric, it gives the relative complexity of process designs, and it is easy to apply(Cardoso et al, 2005).

### 2.4 Design patterns

Design patterns are a mechanism for expressing design structures (Gamma et al, 1994). Design patterns pre-serve design information by capturing the intent behind a design. The use of patterns at architectural level favours the reduction of recurrent reliability modelling work and the understand ability of the reliability model (Laprie, 1996) and allows the designer to reason about fault tolerance and to assign exceptional behaviour responsibilities among components (Lemos, 2006).This allows predicting the effects of particular architectural decisions on the reliability of the system (Klein, 1999).

## 3.0 Experiment Design

### 3.1 Setting of the experiment design

The experiment was set-up to validate the method (MSRAAL) in **section 2.2** which was developed in this research for maintenance risk assessment at the architecture level. For the experiment to validate the risk assessment method, it tests the hypotheses **H1 and H2 in section 1.1**. The experiment measures the complexity of the implementation mechanisms for Internet Protocol (IP) address validation process which are discussed in sections 5.2 and 5.3. IP address implementation mechanism was selected for validation because the internet architecture allows for multiple independent and loosely-coupled component implementation mechanisms due to the fact that different operators and vendors may choose to deploy/develop different mechanisms to achieve the same end, and there need to be different mechanisms to solve problems at different places in a networked environment. The experiment complexity measure (value) is useful for risk assessment of the architecture components/modules which will be either upgraded or redesigned depending on the complexity value of the implementation module under consideration. The experiment is conducted on the Source Address Validation Architecture (SAVA); this is because the SAVA design principles (Wu J, 2007) as discussed below are important in software maintainability.

i. **Performance**

   Deployment of SAVA should not place unreasonable stress on network infrastructure components.

ii. **Scaling**

   SAVA must be capable of scaling to the size of the global Internet.

iii. **Multiple-Fence Solution**

   SAVA should support hierarchical multi-fence solutions to provide different granularities of authenticity of source IP address.

iv. **Loose Components Coupling**

   SAVA should allow for different providers to use different solutions, and the coupling of components at different levels of granularity of authenticity should be loose enough to allow component substitution.

v. **Incrementally Deployable**

   SAVA should show its benefit even if it is deployed only in part of the Internet. If there is no benefit for partial deployment, it is hard to start.

vi. **Benefit to Operator**

   The mechanism should have direct benefit to the party who makes investment on the deployment of the mechanism. Otherwise there is not enough incentive for the global deployment.

### 3.1.1 The experiment

A software prototype for the experiment was designed in such a way to enable the user to input an IP address, the program validates the value input to determine whether the value input for the IP address is valid. The software prototype is implemented at the local subnet level using two different mechanisms i.e. design pattern and the modular approaches (The option buttons provided on the interface determines which implementation mechanism executes when the user clicks the command button **Get IP**). The option button labeled *Design pattern* when selected will trigger the design pattern implementation mechanism to execute which is used for hypothesis H1 testing and the option button labeled *Modular when selected will* trigger the modular implementation mechanism which is used for testing hypothesis H2. The user interfaces for the experiment is as shown in Figure 2 below.

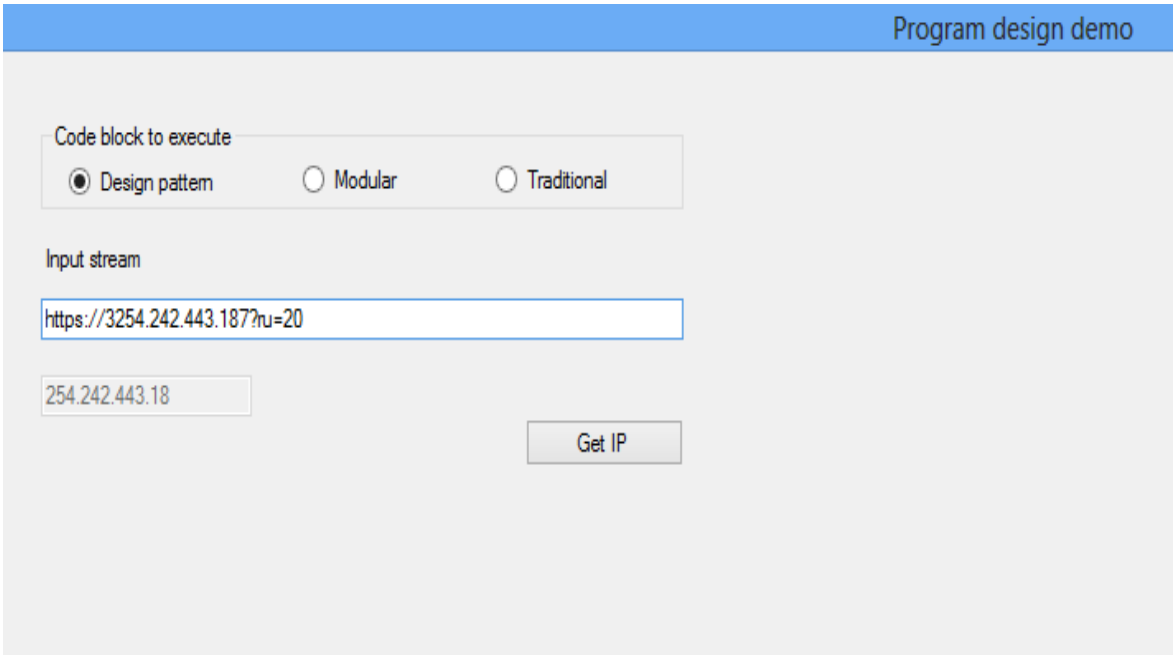



Figure 1 interface for the experiment prototype program demo

## 4.0 Analysis of the results from the experiment

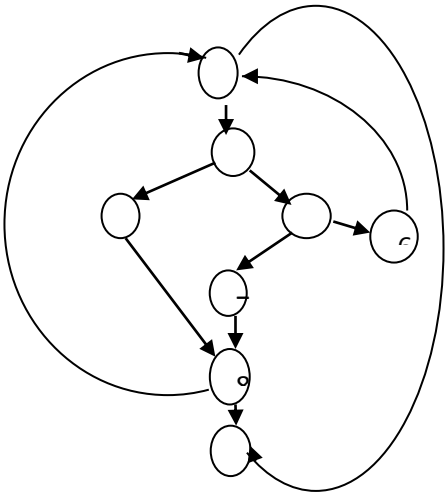
### 4.1 Hypothesis Testing (H1): The modular implementation mechanism code complexity analysis

The modular implementation mechanism was designed to test H1 and it had four interdependent functions to validate the IP address. The cyclomatic complexity for each function is calculated using the control-flow graph by MCcabe(1976) as given below.

```
private int firstIpIndex(string input)
{
    int index = 0;
    while (index < input.Length)
    {
        if (isDigit(input[index]) && index + 13 < input.Length)
        {
            if (input[index + 3] == 46 && 46 == input[index + 7] && 46 == input[index + 11] && isDigit(input[index + 13]))
            {
                return index;
            }
            else
            {
                index++;
            }
        }
        else
        {
            index++;
        }
    }
    return index;// return index corresponding to end of string position
}
```

Figure 2 Function firstIPIndex()



MCC = E-N +X

N= 9

E= 12

X= 2

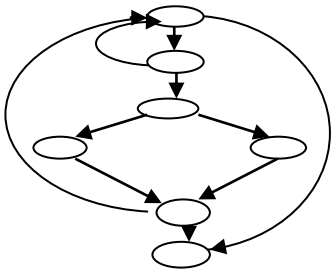
**MCC= 12-9+2 = 5**

Figure 3 control-flow graph of firstIpIndex()

```
private int getCharSet(string input, int index, out string ip, int loop)
{
    ip = String.Empty;
    if (input.Length < index) return index;

    int count = 0;
    StringBuilder sb = new StringBuilder();
    while (count < loop)
    {
        sb.Append(input[index + count]);
        count++;

        if (count < loop && !isDigit(input[index + count]))
            break;
    }

    if (sb.Length < loop)
        sb.Length = 0;
    else
        ip = sb.ToString();

    return index + count;
}
```

Figure 4 Function getCharSet()



**MCC = E-N +X**

N= 7

E= 10

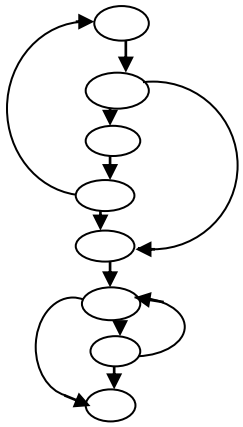
X= 2**MCC= 10-7+2 = 5**

Figure 5 control-flow graph of getCharSet()

```
private int getIpAddress(string input, int index, out string address)
{
    address = string.Empty;
    int groups = 0;

    StringBuilder sb = new StringBuilder();
    while (groups++ < 3)
    {
        index = getCharSet(input, index, out address, 3);
        if (address.Length == 0)
            break;

        if (index < input.Length && input[index] == 46)
        {
            sb.AppendFormat("{0}{1}", address, input[index]);
            index++;
        }
    }

    if (sb.Length == 12)
    {
        address = string.Empty;
        index = getCharSet(input, index, out address, 2);
        if (address.Length == 0)
        {
            return index;
        }
        sb.Append(address);
        address = sb.ToString();
    }
    else
        address = string.Empty;

    return index;
}
```

Figure 6 Function getIpAddress()



MCC= E-N +X

N= 8

E= 11

X=3

**MCC= 11-8+3 = 6**

Figure 7 control-flow graph of getIpAddress()

```
private bool isDigit(char ch)
{
    return ch <= 57 && 48 <= ch;
}
```

Figure 8 Function isDigit()

### 4.2 Hypothesis Testing (H2): The design pattern implementation mechanism code complexity analysis

The design pattern approach tests hypothesis 2. The design patterns implementation mechanism for the IP address validation was implemented by the code as given in Figure 17.

```
if(radioButton1.Checked)
{
    Regex rgx = new Regex(@"(\d{3}.){3}\d{2}");
    textBox2.Text = rgx.Match(textBox1.Text).ToString();
}
```

Figure 9 Design pattern implementation approach

MCC value for the object *rgx* in Figure 9 above is 1. Explanation; the object tests only one condition i.e. it matches its values i.e. input via a textBox2 with criteria provided at the object's declaration to determine the validity of an IP input stream.

### 4.3 Results from the analysis

The modular IP validation mechanism as implemented in the experiment has a cyclomatic complexity value of 17 i.e. (the MCC of *firstIpIndex() = 5 + MCC of getCharSet() = 5 + MCC getIpAddress() =6 + MCC isDigit = 1 thus the total MCC for the modular implementation approach is equal to 17).* The results are as demonstrated by the control flow graphs as per figure 3, figure 5, figure 7 and figure 8 respectively.

The design pattern IP validation mechanism as implemented in the experiment has a cyclomatic complexity value of 1. The result is as demonstrated in figure 9.

### 5.0 Discussion of results

Unnecessary complexity complicates maintenance, since the extra logic is misleading for instance when splitting a control dependency across a module boundary as in the cases of function *firstIpIndex()* in Figure 2 calling function *isDigit()* in Figure 8 and function *getIpAddress()* in Figure 6 calling function *getCharSet()* in Figure 4, there is the risk of introducing control coupling between the modules and limiting the cohesion of each module. Furthermore it is evident from the results that modular implementation approach for IP validation has a higher MCC value of 17 as compared to the design pattern implementation approach which has a MCC value of 1 consequently, the modular implementation is more complex and risky to maintain since it has a very high MCC value of 17 as compared to the design pattern implementation which has a low MCC value of 1.

From the literature review conducted in this research, section2.1, it is revealed that there exist a method (Architecture-Level Prediction of Software Maintenance)ALPSM which provides

comprehensive process support for maintenance risk assessment at the architecture level. The ALPSM does not provide mechanisms to address the risks that are associated with the maintenance changes; this method is used to estimate the efforts which could be expended during the software maintenance process at the architecture level; this method is however insufficient since it does not address the aspect of risk assessment of the software implementation mechanisms used at the architecture level. Risk assessment on the implementation mechanism can be done using the McCabe's (1976) method which measures the code cyclomatic complexity value of an implementation mechanism. Those modules with a MCC value of more than 50, should be redesigned and re-implemented, this is as per ISO/IEC-9126 standard. Risk assessment on architecture implementation mechanism will be of significance since it will enable the software maintainers to calculate the MCC value of a module so as to determine how risky it is before they can proceed to estimate the amount of effort to be expended. This will be very critical in decision making as to whether the module under consideration should be upgraded or redesigned for implementation depending on the module's MCC value.

In testing the hypotheses listed in section 1.1; it is established from the experiment results analysis in section 4.3 that larger modules with more functionality have a higher MCC value of 17 which implies that they will be more difficult and risky to maintain. In H2 testing the MCC value of 1 on the design pattern implementation mechanism is much lower than the modular implementation mechanism of MCC value of 17. (Gamma, 1994) explains that design patterns are a capable technique for achieving widespread reuse of software architectures. The design patterns implementation mechanism captures the static and dynamic structures and collaborations of components in successful solutions to problems that arise when building software in various domains thus, patterns facilitate reuse of software architecture, even when other forms of reuse are infeasible (e.g., due to fundamental differences in operating system features (Schmidt,1995).

**6.0 Conclusions and Future Work**

Most software products undergo maintenance for a number of reasons. Research has established that many companies spend approximately 65% of their software budget on maintenance. It is for this reason that this research attempted to seek the means of reducing the high costs on software maintenance. The contribution of this research is the extension of the Architecture level prediction software maintenance method ALPSM to provide for risk assessment at the architecture level analysis during maintenance. The ALPSM is insufficient since it does not address the aspect of risk assessment of the software implementation mechanisms used at the architecture level.

Risk assessment on architecture implementation mechanism is of much significance since it enables the software maintainers to know the MCC value of the module under maintenance thus establish its maintainability risk level; this will be very critical in decision making as to whether the module under consideration should be upgraded or redesigned for implementation depending on the module's MCC

value it is before the maintainer can proceed to estimate the amount of effort to be expended as given by the ALPSM. This research further established that modules that are implemented using design patterns approach are easily maintainable as compared to those implemented using the modular implementation approach; for instance the module for IP address validation-mechanism implemented using the design pattern approach(in section 4.2) has a Cyclomatic complexity value of 1 which is much lower as compared to that of IP address validation-mechanism implemented using the modular approach(in section 4.1) which has a Cyclomatic complexity of 17. A maintenance risk assessment method at the architecture level analysis is proposed in this research, the method was further validated through the experiment. This method can be of greater benefits for monitoring the effects of individual modifications on complexity. A major problem with software maintenance is that it gets out of control.

The method proposed in this research for software maintenance risk assessment at the architecture level needs to be validated further on different software architectures so as to determine its usefulness. Future researches should propose the metrics that should be incorporated to this method to enhance its easy of applicability.

**References**


1. Abdelmoez, W. M., Goseva-Popstojanova, K., & Ammar, H. H. (2006). Methodology for maintainability-based risk assessment. In *Reliability and Maintainability Symposium, 2006. RAMS'06. Annual* (pp. 337-342). IEEE.
2. Bass, L., Clements, P., & Kazman, R. (2003). *Software architecture in practice*. Addison-Wesley Professional.
3. Bengtsson, P., & Bosch, J. (1999). Architecture level prediction of software maintenance. In *Software Maintenance and Reengineering, 1999. Proceedings of the Third European Conference on* (pp. 139-147). IEEE.
4. Emery, D., & Hilliard, R. (2008). Updating IEEE 1471: architecture frameworks and other topics. In *Software Architecture, 2008. WICSA 2008. Seventh Working IEEE/IFIP Conference on* (pp. 303-306). IEEE.
5. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design patterns: elements of reusable object-oriented software*. Pearson Education.
6. Garlan, D. (2000). Software architecture: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering* (pp. 91-101). ACM.
7. McCabe T. J. and Bulter C.W.(1976). A complexity measure. IEEE Transactions on Software Engineering, 2(4): 308-320.
8. R. Tombe, S. Kimani and G. Okeyo. Method for Software Maintenance Risk Assessment at the Architecture Level. JIARM Journal, 2(1): 290-309, 2014.

9. Sant'Anna, C., Figueiredo, E., Garcia, A., & Lucena, C. (2007). On the Modularity Assessment of Software Architectures: Do my architectural concerns count. In *Proc. International Workshop on Aspects in Architecture Descriptions (AARCH. 07), AOSD* (Vol. 7).
10. Sommerville, I. (2008). Construction by configuration: Challenges for software engineering research and practice. In *Software Engineering, 2008. ASWEC 2008. 19th Australian Conference on* (pp. 3-12). IEEE.
11. Ward W., (1989). Software defect prevention using McCabe's complexity metric. Hewlett Packard Journal, 40(2):64–69, 1989.
12. Wu, J., Ren, G., & Li, X. (2007, October). Source address validation: Architecture and protocol design. In *Network Protocols, 2007. ICNP 2007. IEEE International Conference on* (pp. 276-283). IEEE.