Quality Prediction of Open Source Software for e-Governance Project

Chethan Venkatesh[1]*, Manish Kumar[1], D Evangelin Geetha[1], T.V.Suresh Kumar[1]

ABSTRACT

The government organizations in Africa, Asia and South America are adopting OSS solutions because of its low cost and availability of cheap skilled programmers in their countries to customize the software to their needs. In a study conducted to track the usage of OSS, it was found that OSS is extensively promoted in countries like India, China and Taiwan followed by South Korea, Malaysia, Singapore and Thailand in Asia. In South America, Mexico, Brazil and Argentina are encouraging the use of OSS in all sectors of the government whereas in Africa, South Africa is extensively reaping the benefits of OSS followed by Kenya, Ghana and Nigeria (Noronha, 2003).Many developing countries have a large pool of easily available skilled personnel who can modify the source codes of an OSS to meet the specific needs of the government organizations. The various types of OSS prove to be extremely useful in promoting e-government. In this paper our objective is to propose a methodology which can be used to analyze the existing quality of the OSS as well as to predict the quality of the OSS after customization.

Keywords: OSS/FS - Opens Source Software/Free Software, FLOSS - Free/Libre/Open Source, Software, FOSS - Free Open Source Software, GRAM - Generally Recognized as Mature, GRAS - Generally Recognized as Safe

1. Introduction

Open Source Software / Free Software (OSS/FS) has risen to great prominence. Briefly, OSS/FS programs are programs whose licenses give users the freedom to run the program for any purpose, to study and modify the program, and to redistribute copies of either the original or modified program (without having to pay royalties to previous developers). Many quantitative studies have shown that, in many cases, using OSS/FS programs is a reasonable or even superior approach compared to their proprietary competition. OSS/FS programs are also called FLOSS, FOSS, and libre programs. There are number of Quality Standards are available for Software Quality assessment like ISO-9126, IEEE – 1061 etc. It's the choice of Quality expert to use any methodology to evaluate the quality of software but we recommend that if the evaluation process is using quantitative approach for each and every quality attributes, it will be more appreciable. In this paper we are proposing one small template to evaluate the quality of software using a weighted quantity approach. It's not necessary that Quality Assurance people should use only this approach, but they can use the other approach also in which they are more confident. Our objective is to select the OSS based on the existing quality of the software. Since the software may not be developed

[1] Department of MCA, M.S. Ramaiah Institute of Technology, MSR Nagar, Bangalore-560054 (Karnataka)
* *Corresponding Author*: (E-mail: chethu4u@gmail.com, Telephone: +91 9945178821)

specifically for some particular Govt. Department or organization, so definitely it will require some customization as per the requirement. This customization may change the existing quality of software, so before start doing the changing in existing OSS, our aim is to predict that after customization of the software whether the quality after changes in software will improve or degrade. Depending on the analysis if we see that the quality of the software will be equal to or more than the existing quality of the software, we should adopt the existing OSS software for customization instead of going for the proprietary or new software development.

In the following section we are defining the attributes which we will use to analyze the quality of existing OSS. Based on which we adopt the OSS for customization.

2. Identify Candidates

The first step is to find out what our options are. We should use a combination of techniques to make sure we don't miss something important. An obvious way is to ask friends and co-workers, particularly if they also need or have used such a program. If they have experience with it, ask for their critique; this will be useful as input for the next step, obtaining reviews. Look at lists of OSS/FS programs, including any list of "generally recognized as mature" (GRAM) or "generally recognized as safe" (GRAS) OSS/FS programs. After all, some OSS/FS products are so well-known that it would a terrible mistake to not consider them. For example, anyone who needed a web server and failed to at least consider Apache would be making a terrible mistake; Apache is the market leader and is extremely capable.

3. Read Existing Reviews

After identifying the options, read existing evaluations about the alternatives. It's far more efficient to first learn about a program's strengths and weaknesses from a few reviews than to try to discern that information just from project websites. The simplest way to find these reviews is to use a search engine (like Google) and search for an article containing the names of all the candidates we have identified. Also, search for web sites that try to cover that market or functional area, and see if they've published reviews. In the process, we may even identify plausible candidates we missed earlier.

An important though indirect "review" of a product is the product's popularity, also known as market share. Generally we should always try to include the most popular products in any evaluation. Products with large market share are likely to be sufficient for many needs, are often easier to support and interoperate, and so on. OSS/FS projects are easier to sustain once they have many users; many developers are originally users, so if a small percentage of users become developers, having more users often translates into having more developers. Also, developers do not want their work wasted, so they will want to work with projects perceived to be successful. Conversely, a product rapidly losing market share has a greater risk, because presumably people are leaving it for a reason (be sure to consider whatever its replacement is!).

An interesting indirect measure of a product is whether or not it's included in "picky" Linux distributions. Some distributions, such as Red Hat Linux, intentionally try to keep the number of components low to reduce the number of CD-ROMs in their distribution, and evaluate products first to see which ones to include. Thus, if the product is included, it's likely to be one of the best OSS/FS products available, because its inclusion reflects an evaluation by someone else.

4. Briefly Compare the Leading Programs' Attributes to Your Needs

Once we have read other reviews and identified the leading OSS/FS contenders, we can begin to briefly examine them to see which best meet our needs. The goal is to prepare the list of realistic alternatives to a few "most likely" candidates. Note that we need to do this after reading a few reviews, because the reviews may have identified some important attributes we might have forgotten or not realized were important. This

doesn't need to be a lengthy process; we can often quickly eliminate all but a few candidates. The first step is to find the OSS/FS project's web site. Practically every OSS/FS project has a project web site; by this point we should have addresses of those web sites, but if not, a search engine should easily find them. An OSS/FS project's web site doesn't just provide a copy of its OSS/FS program; it also provides a wealth of information that we can use to evaluate the program it's created. For example, project web sites typically host a brief description of the project, a Frequently Asked Questions (FAQ) list, project documentation, web links to related/competing projects, mailing lists for developers and users to discuss the program or project, and so on

Next, we can evaluate the project and its program on a number of important attributes. Important attributes include functionality, reliability, usability, efficiency, maintenance, and Portability. The benefits, drawbacks, and risks of using a program can be determined from examining these attributes. The attributes are the same as with proprietary software, of course, but the way we should evaluate them with OSS/FS is often different. In particular, because the project and code is completely exposed to the world, we can (and should!) take advantage of this information during evaluation. Each of these will be discussed below; if there are other attributes that are important to us, by all means examine those too.

4.1 Functionality

One of the most important questions is also the simplest: Does the program do what we want it to do? It's often useful to write down at least a brief list of the functions that are important to us. Many project web sites provide a brief, easily-accessible description of the current capabilities of their program; we will have to look at this first. For more information, we can usually read the project's Frequently Asked Questions (FAQ) list and the program documentation. The specific functions that we need obviously depend on the kind of program and our specific needs. However, there are also some general functional issues that apply to all programs. In particular, we should consider how well it integrates and is compatible with existing components we have. If there are relevant standards, does the program support them? If we exchange data with others using them, how well does it do so? For example, MOXIE: Microsoft Office - Linux Interoperability Experiment downloaded a set of representative files in Microsoft Office format, and then compared how well different programs handled them.

We should also consider what hardware, operating systems, and related programs it requires - will they be acceptable for us (do we have them or are we willing to get them)? The issue of operating system requirements is particularly important for organizations that only have Microsoft Windows systems. This is because many OSS/FS programs are only available for GNU/Linux or Unix. In some cases the program can be quickly ported to Windows, but porting is often time-consuming and the products often don't work as well -- typically because Windows does not support some important Unix/Linux features (such as the low-level "fork" capability) or because Windows works significantly differently (e.g., its graphical user interface). Sometimes, particularly with server applications, it may be much better to get an inexpensive computer and install GNU/Linux, FreeBSD, or OpenBSD to use an OSS/FS program than trying to port it to Windows. Today's computers and OSS/FS operating systems are so inexpensive that it's often cheaper to buy special-purpose computers for a task than to try to change the application to run on a different operating system. A positive side-effect is that using a special-purpose computer usually improves security significantly, because we can remove all services from the computer that aren't necessary for its specific task. We can often control server applications using web browsers or remote terminals, so in most cases we can have different operating systems for the desktop and the server application.

4.2 Reliability

Reliability measures how often the program works and produces the appropriate answers; a very similar measure is *availability*. Reliability is difficult to measure, and strongly depends on how the program is used. Problem reports are not necessarily a sign of poor reliability - people often complain about highly

reliable programs, because their high reliability often leads both customers and engineers to extremely high expectations. Indeed, the best way to measure reliability is to try it on a "real" work load, as discussed later. Still, information is often available that may help gauge a program's likely reliability.

In particular, a mature program is far more likely to be reliable. The project's web site itself is likely to try to describe the program's maturity; if the project declares that the program is not ready for end-users, they're usually right. To be fair, some developers are perfectionists and are never willing to admit that a program really is mature. Another *very* good sign is the presence of a test suite that is used *at least* before official releases, and preferably constantly (e.g., daily or with each check-in). The test suite should be included with the source and/or binary release of the program; if they aren't, there's the risk that they aren't getting maintained as well as they need to be. Test suites can help prevent common problems from re-occurring, and can reduce the risk of a change causing an undesirable side-effect. For this reason, test suites are especially valuable if we decide that we want to make changes to the software. But even when we aren't making changes to the software, the presence of a reasonable test suite suggests that the program is more reliable. Take a look at the test suite, to see how well it covers the program's functionality.

4.3 Usability

Usability measures the quality of the human-machine interface for its intended user. A highly useable program is easier to learn and easier to use. Some programs (typically computer libraries) are intended only for use by other programs and not directly by users at all. In that case, it will typically have an application programmer interface (API) and we should measure how easily programmers can use it. Generally, an API should make the simple things simple, and the hard things possible. One way to get a measure of this is to look for sample fragments of code that use the API, to see how easy it is to use.

For applications intended for direct use by users, there are basically two kinds of human-machine interfaces used by most of today's programs: a command-line interface and a graphical user interface (GUI). These kinds are not mutually exclusive; many programs have both. Command line interfaces are easier to control using programs (e.g., using scripts), so many programmers and system administrators prefer applications that have a command line interface available. So, if the application will need to be controlled by programs sometimes, it is a significant advantage if it has a command line interface. There are alternative user interfaces for special purposes. For example, there are programs (typically older programs) which use character screens to create a text-based GUI-like interface (these are sometimes called "curses" programs, named after a library often used to build such programs). However, for most applications intended for direct use by humans, today's users want a GUI; GUIs are much easier to use for most users. In most circumstances, it will be the program's GUI that you'll be evaluating.

It's important to note that many OSS/FS programs are intentionally designed into at least two parts: an "engine" that does the work, and a GUI that lets users control the engine through a familiar point and click interface. This division into two parts is considered an excellent design approach; it generally improves reliability, and generally makes it easier to enhance one part. Sometimes these parts are even divided into separate projects: The "engine" creators may provide a simple command line interface, but most users are supposed to use one of the available GUIs available from another project. Thus, it can be misleading if you are looking at an OSS/FS project that only creates the engine - be sure to include the project that manages the GUI, if that happens to be a separate sister project.

In many cases an OSS/FS user interface is implemented using a web browser. This actually has a number of advantages: usually the user can use nearly any operating system or web browser, users don't need to spend time installing the application, and users will already be familiar with how their web browser works (simplifying training). However, web interfaces can be good or bad, so it's still necessary to evaluate the interface's usability.

4.4 Efficiency

Have functions been optimized for speed? Have repeatedly used blocks of code been formed into sub-routines? Checked for any memory leak, overflow? The amount of computing resources and code required by a program to perform its function.Performance bottlenecks can be identified, and the need for performance tuning can then usually be localized in a small number of performance-critical components. Components can be internally optimized to improve performance, without affecting their specification; components can be moved between platforms to improve performance, without affecting the functionality or usability of the application. ISO/IEC 9126 identifies efficiency as "A set of attributes that bear on the relationship between the software's performance and the amount of resources used under stated conditions". In fact the term efficiency refers to the time and resource behavior of the software. There are, of course, some benchmarks from various popular magazines or some web sites, showing benchmark data about the performance, but sometimes the rationale of the benchmarking test is in most times, quite subjective.

4.5 Maintainability

The Maintainability of a system is its aptitude to undergo repair and evolution. The modular structure of a component based solution allows individual components to be replaced easily. Maintainability of a software is highly dependent on the process used to develop it. Assessing the quality of the software maintenance process is done using a software maintenance maturity model. Assessing the maintainability of the software product is done by assessing four different perspectives: its Analyzability, Changeability, Stability and Testability.Maintainability has to do with the easiness of code modification.

According to the ISO 9126 (International Organization for Standardization, 1991) model, maintainability is "A set of attributes that bear on the effort needed to make specified modifications (which may include corrections, improvements, or adoptions of software to environmental changes and changes in the requirements 22 and functional specifications)". Maintainability of F/OSS projects is a factor that was one of the first to be investigated by the F/OSS literature. This was done mainly because F/OSS development emphasizes on the maintainability of the software released. Making software source code available over the Internet allows developers from all over the world to contribute code, adding new functionality (parallel development) or improving present one and submitting bug fixes to the present release (parallel debugging). A part of these contributions are incorporated into the next release and the loop of release, code submission/bug fixing, incorporation of the submitted code into the current and new release is continued. This circular manner of F/OSS development implies essentially a series of frequent maintenance efforts for debugging existing functionality and adding new one to the system. These two forms of maintenance are known as corrective and perfective maintenance respectively. In another study, the maintainability of the source of an open source product is compared directly with a closed source one (Samoladas et al., 2003).

In that study, the Maintainability Index (MI) proposed by Carnegie Mellon's SEI (Software Engineering Institute, 2002) was used in order to gain insight into the evolution of the maintainability of five F/OSS systems. The reason behind the use of 24 the MI stem from concurrence by both the authors and McConnell's that F/OSS should conform to such standards and metrics. The results of the study show that F/OSS maintainability is no worse than that of closed source software. In direct comparison, F/OSS was found to be better than closed source software

4.6 Portability

Does the program depend upon system or library routines unique to a particular installation? Have machine-dependent statements been flagged and commented? Has dependency on internal bit representation of alphanumeric or special characters been avoided? The effort required to transfer the program from one hardware/software system environment to another. The specification of a component is platform-independent. So a component can be quickly rebuilt for a new platform, without affecting any

other component. For example Java projects may need to be tested on different operating systems and application servers. Web applications may need to be tested in different browsers. Portability in software is an attribute that has to do mainly with platforms and machine dependence. It expresses the ease of transferring an existing system, running on a specific machine, to another machine with a different configuration. ISO/IEC 9126 (International Organization for Standardization, 1991) describes portability as "A set of attributes that bear on the ability of software to be transferred from one environment to another (including the organizational, hardware, or software environment)". From its early days, portability has been a central issue in F/OSS development. Various F/OSS systems have as first priority the ability of their software to be used on platforms with different architectures. Here, we have to stress one important fact, which originates from the nature of F/OSS, and helps portability, namely the availability of the source code of the destination software. If the source code is available, then it is possible for the potential developer to port an existing F/OSS application to a different platform than the one it was originally designed for. Perhaps the most famous F/OSS, the Linux kernel, has been ported to various CPU architectures other than its original one, the x86. In the end, evaluating usability requires hands-on testing.

5. Proposed Evaluation Sheet and Evaluation Methodology

Now after going through all the factors in detail this is the time to quantify these all analysis in a metrics form. We are proposing an analysis metrics to quantify this entire evaluation process.

Table 1: Evaluation Template

		Weighting Factor between 1 to 5	**Maximum Score = (Weighting Factor) x 10**	**Evaluated Score out of 10**	**Weighted Score = (Weighting Factor) x (Evaluated Score out of 10)**
Software Quality Attributes					
I	Functionality				
II	Reliability				
III	Usability				
IV	Efficiency				
V	Maintainability				
VI	Portability				
TOTAL					

5.1 Evaluation Methodology

1. Calculation for Software Quality Attributes

- **(a)** Total Weighting Factor of Software Quality Attributes i.e. $\boldsymbol{WF_I}$ = Sum of weighting factor of all six attributes in Software Quality Attribute.
- **(b)** Total Evaluated Score of Software Quality Attributes i.e. $\boldsymbol{ES_I}$ = Sum of evaluated score of all six attributes in Software Quality Attribute (each attribute score will be measured out of 10)
- **(c)** Total Weighted Score of Software Quality Attributes i.e. $\boldsymbol{WS_I}$ = Sum of weighted score of all six attributes in Software Quality Attribute.

2. Total Weighting Factor i.e. $\boldsymbol{TWF} = \sum_{i=1}^{6} WFi$

3. Total Weighted Score i.e. $\boldsymbol{TWS} = \sum_{i=1}^{6} WSi$

Now we have to calculate the percentage of each Weighting Factor with respect to Total Weighting Factor (***TWF***) i.e.

$$\boldsymbol{PWFi} = \frac{WFi}{TWF} \times 100 \qquad \text{for } 1 \le i \le 6$$

Similarly we have to calculate the percentage of each Weighted Score with respect to Total Weighted Score (***TWS***) i.e.

$$\boldsymbol{PWSi} = \frac{WSi}{TWS} \times 100 \qquad \text{for } 1 \le i \le 6$$

We recommend that Total Weighted Score (TWS) should be at least 70% of Total Max Score (TMS). But only this condition is not sufficient to adopt the software. If you analyze the evaluation process you can see that we have divided the evaluation in six major factors i.e. Functionality, Reliability, Usability, Efficiency, Maintainability, Portability. Total Weighted Score (TWS) is sum of weighted score of all the six major factors mentioned earlier. Similarly Total Max Score (TMS) is also a sum of Max Score of all the six major factors. Since in entire evaluation process Weighting Factor is playing a key role so it's important to consider that the major factor (Functionality, Reliability, Usability, Efficiency, Maintainability, Portability.) whose weighting factor is more should contribute more in a Total Weighted Score. Because sometime it may happen that few major factor is less weighted but they are contributing very high score in Total Weighted Score and the other major factor who's weighting factor is very high are contributing very less in Total Weighted Score. But finally the Total Weighted Score is satisfying the recommended criteria (at least of 70% of Total Max Score (TMS)).So In this case only considering the Total Weighted Score will not give a feasible solution.

So to overcome with this problem we are considering the following condition.

$$\text{ER} = \sum_{i=1}^{6} f_i \quad \text{where} \quad f_i = \begin{cases} 1 & if\,(PWSi < PWFi) \\ 2 & if\,(PWSi = PWFi) \\ 3 & if\,(PWSi > PWFi) \end{cases} \qquad \text{for } 1 \le i \le 6$$

Here ER is a final Evaluated Result and we suggest that ER value must be equal to or greater than 10.Using the given templates we can evaluate the quality of both proprietary as well as OSS. Finally ER value will give some effective result which can be used for taking the decision for considering the proprietary or OSS for the better quality of project.

Since the OSS whatever we analyzed using the above described methodology may not be developed specifically for some particular Govt. Department or organization, so definitely it will require some customization as per the requirement. This customization may change the existing quality of software, so before start doing the changing in existing OSS, our aim is to predict that after customization of the software whether the quality after changes in software will improve or degrade. Depending on the analysis if we see that the quality of the software will be equal to or more than the existing quality of the software, we should adopt the existing OSS software for customization instead of going for the proprietary or new software development. So in the next section we are proposing a method for the early prediction of OSS quality i.e. the prediction of the OSS quality after customization.

6. A Software Quality Prediction Model Based on FALCON

In this section, we propose a software quality prediction model based on FALCON, a specific type of FNN. There exist many factors that have impact on the quality of target software product, therefore, the relationship between these factors and software quality attributes is a complex, nonlinear and multivariate one. Naturally, FALCON could be a good candidate for software quality prediction modeling. Moreover, due to the fusion of ANN and FL, FALCON possesses the advantage of natural language description of FL as well as the learning properties of ANN. This motivates our proposition of a software quality prediction model using FALCON. The ISO 9126, IEEE 106 standard for information technology and proposed method provides a framework for the evaluation of software quality, which defines six product quality attributes, i.e., functionality, reliability, usability, efficiency, maintainability and portability. Nevertheless, when conducting software quality prediction, one should choose those quality attributes of interest and of major importance as the output variables of the model, rather than to stringently use all the six quality attributes as output variables, since the increase of the number of output variables will considerably increase the complexity of the model as well as the time for model training. In our methodology, all output variables of the software quality prediction model are defined as variables which can only take values in [1,2,3] for which a value 3 indicates a higher value of quality attribute (e.g., higher portability) and a value 1 indicates a lower value of quality attribute (e.g., lower portability). The input variables of the software quality prediction model are factors which are considered to have impact on the target software's quality attributes, including software product metrics, development process characteristics, operation conditions, etc. From a practical point of view, only those factors which, for a specific software project, have certain available measurements and have significant impact on quality attributes should be used as input variables. Figure 1 [13] illustrates a software quality prediction model constructed based on FALCON. In this example, we choose three levels (Low, Medium and High) for each input and output variable. As there are three input nodes at layer 1 and each of them has three term nodes, there is 3×3×3, namely 27 nodes in layer 3, each of which represents a possible fuzzy logic rule. The term nodes in layer 2 and layer 4 are realized by Gaussian fuzzy membership functions. The links between layer 3 and layer 4 are fully connected before training. Each link has a weight to represent the strength of the existence of the corresponding rule consequence. For each of the input variables, its value to be used in this model is normalized to [-1, 1].

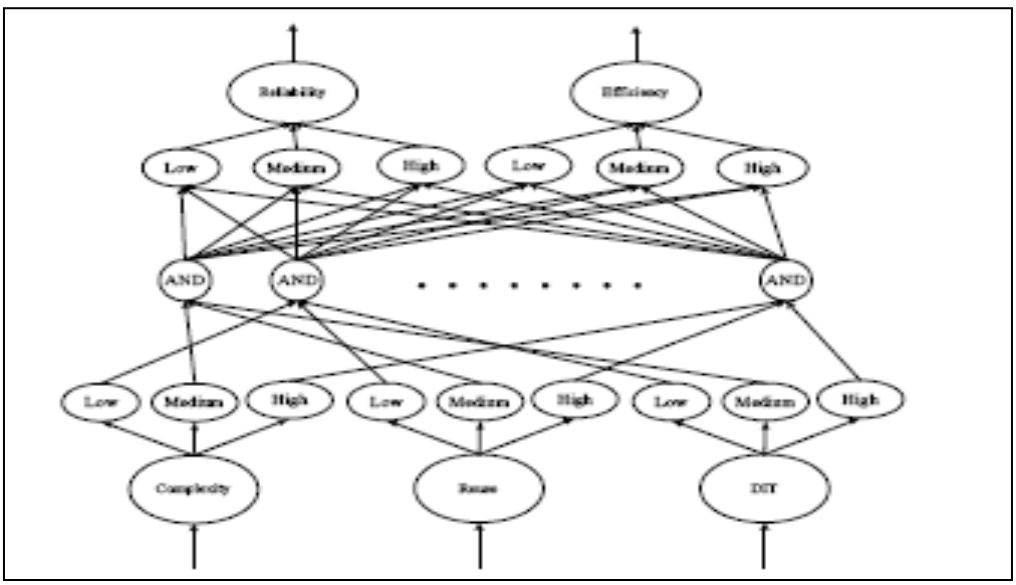

Figure 2: A Software Quality Prediction Model based on FALCON (before training)

In summary, the proposed FALCON-based model has some features that are quite favorable in software quality prediction:

- It is a multi-input-multi-output model, thus it can describe the complex relationship between target software's quality attributes and all those affecting factors. On the one hand, the multi-input feature makes the model able to take into account most factors which have impact on quality attributes; on the other hand, the multi-output feature makes the model able to predict several quality attributes of interest.
- In a software development project, the information that can be obtained includes objective data and knowledge/experiences from experts or from similar projects (which may be in form of a rule such as "if the number of classes is too big, then the reliability and maintainability of the target software will both be low"). Therefore, a realistic software quality prediction model should have the capability of handling both types of information, which is especially crucial to early software quality prediction. The FALCON-based model, due to its nature of fusion of artificial neural networks and fuzzy logic, has the capability of dealing with both objective data and people's knowledge/experiences. For knowledge given in form of rules, it can be used by the model in the training phase by directly entering the model from the second layer.
- The model has much flexibility. In the early phases of software development process, there is not much

7. Concluding Remarks

Many developing countries have a large pool of easily available skilled personnel who can modify the source codes of an OSS to meet the specific needs of the government organizations. The various types of OSS like compilers, firewalls, networking software, operating systems etc. prove to be extremely useful in promoting e-government. In this case if we can predict the quality of OSS before customization, it will help the organization to take a clear decision about the Proprietary Software and OSS.

References

1. Abiteboul, S.; Leroy X.; Vrdoljak B. (2005) *EDOS: Environment for the Development and Distribution of Open Source Software*, in International Conference on Open Source Software.
2. Bo Yang, Lan Yao, Hong-Zhong Huang. (2007) *Early Software Quality Prediction Based on a Fuzzy Neural Network Model*, ICNC
3. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: (2001) *Introduction to Algorithms.* 2nd edn. MIT Press and McGraw-Hill
4. Diomidis Spinellis (2006). *Code Quality:* The Open Source Perspective. Addison-Wesley, Boston, MA.
5. *Enterprise Management Associates: Get the truth on Linux management.* Online http://www.levanta.com/linuxstudy/, Current Jan 2007 (2006)
6. *Environment for the development and Distribution of Open Source software (EDOS)*, http://www.edos-project.org/.
7. Fielding, R. T. (2000) *Architectural Styles and the Design of Network-based Software Architectures,* PhD Thesis, University of California, Irvine.
8. González-Barahona, J. M.; Conklin, M. and Robles, G. (2006) *Public Data about Software Development*, in International Conference on Open Source Software.
9. Halstead, M.H. (1977): *Elements of software science. Operating and Programming Systems* Series 7
10. *International Organization for Standardization: Software Engineering - Product Quality - Part 1: Quality Model. ISO,* Geneva, Switzerland, 2001. ISO/IEC 9126-1:2001(E). Online, Current Jan 2007 (2001)
11. McCabe, T.J., Watson, A.H. (1994) *Software complexity. Crosstalk, Journal of Defense Software Engineering* 7(12)
12. Spinellis, D. (2006) *Code Quality: The Open Source Perspective.* Addison-Wesley, Boston, MA
13. Stewart, K. J. and Gosain S. (2001) *An Exploratory Study of Ideology and Trust in Open Source Development Groups,* in International Conference on Information Systems.

About the Authors

Chethan Venkatesh received MCA from VTU and MPhil in computer science from Bharathidasan University. His areas of interest are Computer Networks, Software Engineering, Software Quality & Testing. He is currently working as a lecturer in department of MCA, M S Ramaiah Institute of Technology.

Manish Kumar is a faculty in MCA Department, M.S. Ramaiah Institute of Technology, Bangalore, India. His areas of interest are Cryptography and Network Security, Distributed and Parallel Processing, Mobile Computing, E-Governance and Open Source Software. His specialization is in Distributed Parallel Processing. Before joining teaching profession, he worked on various software development projects for the government and private organizations. He worked on the R&D projects related on theoretical and practical issues about a conceptual framework for e-mail, website and cell phone tracking, which could assist in curbing misuse of information technology and cyber crime. He has published several papers in International and National Conferences.

D Evangelin Geetha received MCA degree from Madurai Kamaraj University, India in 1993. Pursuing Ph.D in Computer Applications from Visvesvaraya Technological University, Belgaum, India. Her areas of interest are software performance engineering, Object Technology, Distributed Systems. She is currently working as a Assistant Professor in MCA Department, M S Ramaiah Institute of Technology, Bangalore. She has published several papers in National, International conferences and Journals.

T V Suresh Kumar is working as a Professor and HOD in Department of MCA, M. S. Ramaiah Institute of Technology. His areas of interest are Software Performance Engineering, Object Technology, Distributed Systems and Reliability Engineering. He worked on various software development and research project of DRDO, CASSA etc. He is visiting faculty in various universities and reputed software development organization. He has published many papers in National, International conferences and Journals.