Software Metrics & Associated Issues A Literature Survey*

Arvind Gopu
Graduate Student, Computer Science & Informatics
Indiana University, Bloomington, IN
agopu [at] cs.indiana.edu

December 19, 2003

Abstract

The importance of Software metrics has grown in the software engineering community, especially in the past two decades with the development of new and improved metrics. Metrics have been used more and more in making quantitative/qualitative decisions as well as in risk assessment and reduction. They give software professionals the ability to evaluate software process. Thus it is crucial to reason any metric that is proposed and validate it using standardized techniques. It is also important to consider the psychological and economic aspects of using software metrics because these metrics would be of no use if they are not used the right way. This paper is an attempt to discuss some of these issues based on a literature survey. Every attempt has been made to make it as concise as possible, but in the process a few details could have been missed. We strongly recommend the reader of this paper to refer the original papers that are cited at various points.

*Towards partial requirement of CSCI B665 - Software Engineering Management

1 Introduction

"What is not measurable, make measurable", the great Galileo Galilei had said. Measurement has always been fundamental to any engineering discipline and software engineering is no exception. This is how Pressman [8], introduces metrics. So what kind of measurement is he talking about? Obviously it should be something that gives us the ability to evaluate software process – the design, the code, the testing, etc. But does it end there? Probably not. Metrics also cover the aspect of evaluating the final software product and a lot more.

Recently, the importance given to software metrics has grown in the software engineering community. These metrics have been used more and more in making quantitative/qualitative decisions as well as in risk assessment and reduction. The past two decades especially have seen the development of new and improved metrics and a lot of related efforts in validating those metrics. There have been a number of publications to these effects. In this paper we have surveyed the literature for some interesting papers on software metrics and validations that have been proposed over them.

In section 2 we discuss some of the basic concepts of software metrics. We go into the question of why we need metrics in the first place and also give a brief description of the various classifications that have been in use in the software engineering community. In section 4 we discuss some recently proposed metrics - ones that are claimed to be better than traditional metrics. We also briefly discuss some special cases where metrics are applied, for example in a database application, in section 5. We mention some important issues related to metrics – the human element being the most important one, in section 6 before concluding with section 7.

2 Basics of Software Metrics

In this section we discuss some basic concepts of software metrics beginning with the definition:

2.1 Defining Software Metrics

The definition of software metrics has taken various forms since its inception. Metrics are quantitative measures that enable software people to gain insight into the efficacy of software process and also pinpoint problem areas [8]. They provide requisite information for quantitative managerial decision making as well as support for risk assessment and reduction [5]. They can be considered an objective mathematical measure of software that is sensitive to difference in software characteristics [14]. According to the IEEE standard glossary of Software Engineering Terms (adapted from [8]), they are "a quantitative measure of the degree to which a system, component, or process possesses a given attribute.".

The term "metric" itself refers to a wide range of activities that are related to measurement in software engineering. These include [5]:

- Quantitative values characterizing properties of code
- Prediction models to predict resource requirements and end-product quality
- Quantitative aspects of quality control, etc.

Almost every metric has one thing in common – the motivation behind it; this could be assessing cost and effort to be put in or assessing the quality of the software. For example, one of the earliest metrics to measure code efficiency – the Lines of Code (LOC) metric has been used in a model as simple as:

$$Effort = f(LOC)$$

There have been various other attempts to use, for example, LOC (and other naive metrics) as a metric to measure aspects such as effort, complexity, etc. It is pretty obvious that with the advent of newer programming languages these models were not going to work because it does not make sense, for instance, to compare LOC values of an assembly language program with a high-level language program. Thus researchers started working on coming up with metrics that were independent of the language used. We address some examples of such metrics in section 4.

2.2 Need for Metrics

Having looked at some basics of software metrics, the next question that arises in many a budding software engineering's mind is why do we need metrics? And if they are indeed useful to their cause? The answer to the latter question in short is yes. Software metrics have been proven to be useful if applied in the right way – this needs to be stressed. Application of wrong methods might lead to failure of a metric but it is not really the metric which is to be blamed! Here we state some arguments as to why we need software metrics:

- Without measuring software process or the quality of an end product, only subjective evaluation is possible – not desirable.
- With robust measurements [8]:
  - Requirements can be assessed better
  - Error prone components can be identified at early stages
  - Quality assurance can be improved
- Predicting resource requirement is another important use of software metrics

2.3 Classification of Metrics

There are many available classifications of software metrics. Most of these are only different in a subtle way. A clear classification list is given in [11] and it is based on the following factors:

1. Complexity
2. Quality
3. Effort/Competency
4. Performance
5. Style (and so on . . .)

There might be subclassifications based on this. For example, complexity is usually sub classified on the following basis:

- Structural
- Logical
- Computational
- Lexical and
- Psychological

More often software metrics is categorized in a much broader sense as:

1. Process metrics and
2. Product metrics.

The former refers to attributes assigned to the process using which the software was constructed. The latter refers to end product from both the user's perspective as well as the developer's. Gaffney has a more detailed and descriptive classification in his work [14].

Process metrics are broadly sub-classified as:

1. Structure metrics: Comes in handy at the initial stages of a software's development. Examples are:
    - Information flow
    - Invocation complexity
    - Review complexity
    - Stability measure.

2. Code Metrics: Usually applicable only to latter stages of the software development cycle. This is considered a drawback but yet it has been proven to be useful for programmers and managers alike, especially when used in combination with structure metrics. We will discuss a bit more on that in section 4. Examples of code metrics are:

   - Lines of Code (LOC)
   - Software Science "effort" measure
   - McCabe's cyclomatic complexity

3 Validation of Software Metrics

With the plethora of metrics proposed it is critical that these metrics are thoroughly validated with the help of past experiences and new test data. There are many views on how this validation should be carried out. For example, Ejiogu [11] suggests that since metrics touch both structured programming and mathematical measure theory, the two need to be combined when validating software metrics. He goes on to expand on this point in the paper. This is discussed in section 4.2. Usually metrics are validated using simple regression or linear rank correlation techniques. These have proved to be effective in a lot of simple cases.

But as the complexity of software grows and also as more and more metrics are proposed, these simple techniques have been looked at, with more skepticism. This has lead to more research in this area of validating software metrics, some of which we will discuss in this paper. The reason why regression based models may fail in validation of software metrics can be explained by Fenton et al's analogy [5]. They argue that regression models lead to misleading results and cite a road accident analogy – just claiming that winter is the best time to drive is flawed if the decision is just based on lower number of accidents; the fact that lesser number of people actually drive in winter conditions is very crucial. We will discuss their proposed technique in a bit more detail as well as talk about one obvious advantage of better validation techniques – the fact that it churns out newer and more effective metrics, in section 4

We will now discuss a novel validation technique proposed by Kafura and Canning [13, 12]. They have based their validation on a realistic software system, something they argue is important. The term resource in the context of this paper (this is actually applicable in most contexts) refers to the combination of component coding time and the number of component errors. Errors are combined with the coding time because it is expected that some one will spend time to fix the errors. The authors first point out that existing validations were always based on linear or rank correlations to figure out how the metrics related to the features of the system. They argue this approach is very limited in its effectiveness because of three reasons:

- Use of single dependent variable means they fail to deal with possible trade-offs between resources. This might mislead us to think the metric

failed though the failure indeed may have had to do with the validation procedure. For example, the validation might relate number of errors with a number of metrics (independent variables) but still might fail to deal with trade-offs that can be made between resources.

- Use of single metric (independent variable) means only certain aspects of software quality is measured. Even in attempts to use more than one metric, the metrics have been from a particular class – structural or code or the like as described in section 2.3 and it is strongly argued that these classes measure different aspects of a software quality.

- Poor correlation between metric and resource might mislead us to think the metric is a poor one if, for example, a component with a low metric value ends up with a lot of errors. This is not necessarily true - it might be the case where a wrong metric might have been used or there might have been poor testing.

The main point to infer is that different classes of metrics measure different aspects of system quality, so a single metric or multiple metrics from a class cannot be successfully used to measure all quality related facets of a system. The authors go on to give a description of the analysis they did with various metrics. Their analysis is based on three considerations which are explained below along with some information on the results of the analysis:

1. The first consideration is if significant differences in metric values imply corresponding differences in errors and/or coding time. And does considering the two factors as a combination lead to a more pronounced relationship between the metrics and the resources? The analysis shows that the answer to the above questions is 'yes'. Tabulated results using a combination of Information flow and Lines of code show that 'easy' components have a small metric value while 'difficult' components have a higher metric value. In this context 'easy' components correspond to ones that take lesser time to code and also carry lesser number of errors whereas 'difficult' components are the ones that take longer to code and carry more errors. The above mentioned trend is much more pronounced when both metrics are considered at the same time.

2. The second consideration is if the metrics can identify trouble maker components – ones which are more error-prone. Again, does using more than one metric have an effect on this? Especially the authors are interested in figuring out if structure metrics can find the trouble making components in a software system so that more importance can be attributed to testing those modules. Standard deviation from mean metric value has been used to demarcate boundaries between outliers, extreme outliers and non-outliers. Outliers are components with a value one standard deviation away. Extreme outliers are components with a value twice the standard deviation away. It was not easy to pick one best metric in this analysis.

The only inferences that could be made are relative – like which metric in each class is best. Also it has been found that using three or four metrics for this aspect of testing is ideal. This is empirical data though without much theoretical backing.

3. The last aspect the authors have considered is the number of false positives if we could call it that. Essentially computing the number of high metric values without an associated error or high coding time is important. Safety factor – the percentage of components which do not fall in the outlier category for any of the metrics under use – is another factor to take into consideration. The decision to use a particular metric also depends on economics of concerned party using the metrics. The authors have some numbers to support their analysis for this aspect as well as the above mentioned ones in the paper [13].

4 Improved Metrics

As mentioned earlier, one of the main drawbacks of metrics is the fact that most metrics address only one aspect of the multifaceted software development process. Thus a traditional metric might fail to take into consideration, for example, the trade off between two resources because its dependent variable is only one of those resources. Researchers have tried various methods to overcome this problem. One of those methods which is more used than any other is combining metrics from various classes to give a more general picture.

4.1 Causal Models for Metrics

To illustrate how a single metric can be proved to be ineffective and how smart use of combination of the same metrics can produce better results we discuss a proposition by Fenton et al [5]. We had made a brief mention of this in the previous section, now we will give a bit more detail. Just to recollect, the term resource especially in the context of this paper refers to the combination of component coding time and the number of component errors since the latter is expected to be fixed by some one with expenditure of time.

The authors argue that most metric approaches use regression based models for estimating resource requirements as well as error prediction. and do not give much insight in terms of how risk can be reduced. The authors suggest that factors like causality, uncertainty, etc should be considered and to achieve that they recommend use of Bayesian nets and such in modeling the metric approach.

So why does a regressed based model not suffice? To answer that question, a useful and a common analogy – again briefly mentioned in the previous section – to consider is the correlation between month of the year and the number of fatalities due to road accidents. Unless cause and effect is looked at with the correct perspective, it is possible to end up with the awfully wrong conclusion that winter is the best time to drive since figures would indicate least number

of accidents in winter. Conditions like road condition, weather, etc need to be taken into consideration which a normal model would not. This is the pith of the authors' argument. There is a basic model shown (pictorial) in the paper [5] which relates size and additional factors to the effort put in/quality produced.

The use of pre-release defects as an indicator of quality is also questioned. Knowing that there were a large number of defects during the coding stage does not mean there will be a lot of bugs in the post release version too. To emphasize this, an analogy can be used – predicting that a person who has eaten a heavy lunch will also eat a heavy dinner, is this a sensible inference? A thorough analysis shows that indeed fault prone modules in the post release version were ones that revealed no or very small number of pre-release faults, i.e., they could have been validated lesser. This leads us to a more intuitive conclusion than the previous argument. Also this indicates that amount of testing is another factor which should be built in to quality/error predicting models. Similar is the case with operational usage and resource constraints associated with any project. Limited historical data availability is another constraint basic regression models suffer from though it is not clear how newer models can overcome it.

The answer to many of the above mentioned problems seems to be use of causal models, for example Bayesian Belief Nets (BBN). Another method which is indicated to be closely related is the Process Simulation method. A picture of a typical application of BBNs to software engineering and metrics is shown in the paper. A typical BBN is a network which has an associated set of probability tables. Each node in the network represent an uncertain variable while each edge represents a causal relationship. The above mentioned probabilities would be used to indicate the probability a node is in a particular state. It is argued that causal models like ones which use BBNs can:

- Handle diverse process and product variables
- Can correlate with empirical evidence as well as expert predictions
- Uncertainty and incomplete information
- Avoid introduction of additional overhead or at least keep it at a minimum

Those advantages aside, the method involves building the BBN probability tables using both empirical data and expert predictions. The authors claim to have developed tools to compute these probabilities in huge scales very quickly. The resulting BBN as the one shown in the paper [5] will contain both variables with empirical/expert data and variables which are being predicted – for example, post release errors. New evidence is periodically used to update the variables representing known data. The authors argue that use of BBN enables them to compute all probabilities including the unknown ones (in terms of empirical data), this is in fact claimed to be one of the main advantages of their method. They have a figure representing their model and also one showing a sample BBN and how their model worked in evaluating unknown variables to have values as expected. In the end the authors also mention the need for a

simple interface between complicated metric models and people who manage software – managers would not want to figure out how BBNs work, they would rather prefer to get results out of a partial black box, partial so that they are kept interested.

4.2 Mathematical Measure Theory for Metrics

In this section we discuss the views put forward by Ejiogu [11] in proposing a unified theory for software metrics based on mathematical measure theory. There have other researchers who have suggested the use of formal frameworks for metrics and pointed out the advantage of doing so. For example, see Purao et al [1]. Coming back to Ejiogu's work, the main argument in his paper is that mathematical measure theory is directly applicable to metrics. The fact that structured programming is analogous to tree design is cited and it is suggested that the tree structure be advanced as a formal space for defining software metrics.

Also use of count of components instead of the traditional lines of code metric is suggested to get a better sense of size of software. Each component can be represented by a node in a tree structure with the usual parent-child relationships. The author also argues that factors like software behavior, feedback effects of using metrics should not be ignored. Another point that the author makes is about the myth that complexity does not depend on size of a module; it might be the case at times, but usually it is easy to prove that structural complexity has a direct effect on the size of a component. This should be taken into consideration.

Moving on to the propose metric itself, the author argues that unlike usual metrics which do not satisfy basic mathematical measure rules, the proposed one satisfies the null condition as well as the conditions of monotonicity and countable sub additivity. Thus if there is a measure μ, then:

- For an empty set ϕ, it should satisfy $\mu(\phi) = 0$
- For A $\subset$ B, $\mu(A) \leq \mu(B)$
- $\mu(\cup_i A_i) \leq \Sigma_i \mu(A_i)$

The author has made use of the fact that structured programming allows the use of tree structure to present a unified theory for software metrics, one that satisfies the above mentioned conditions. Notions like union, intersection, complements are inherently built-in in trees. To put it simply, a software program refined into simpler sub programs till its monadic function limit (leaf nodes in the tree) can be represented as a tree structure. This phenomenon of refinement of a problem is referred to as *nesting*. There is another phenomenon called *twinning* in which the nodes split into two or more child nodes – the factor of splitting being twin factor. The above mentioned phenomena correspond to data structure terms – stacks and bundles respectively. Deeper the nesting, more complex the problem is and similarly larger the bundle, bulkier is the problem.

A numerical metric for structural complexity is defined:

$$S_c = L * R_m * M$$

where
L is the level in the tree
R_m is the root node's twin number and
M is the number of monads

Having defined the terms described above the author goes on to define some terminology relating trees to software programs. For example:

1. Complex or node
2. Height of a node
3. Level of a tree
4. Subtree & Maximal subtree
5. Monad
6. Nesting & Twinning, etc.

Most of these are common tree terminologies; the terms nesting and twinning have been explained previously. Using these terms, the authors finally define metrics that can be used to evaluate software:

- Mass/size (M_x/S_z)
- Structural complexity as defined previously
- Degree of structural complexity
- Degree of structuredness
- Degree of maintainability, etc

Expanding on these terms is out of the scope of this paper.

5 Special Cases

In this section we discuss some specialized software metrics applied only in certain domains/methodologies. For example, Object Oriented Programming paradigm and its associated design methodologies require a different set of metrics from common ones because of their unique nature. A variety of metrics have also been proposed specifically for use in database applications, software reuse and so forth. We will briefly discuss Object Oriented software metrics and a metric for database applications in the following sub-sections. For more details, we suggest the reader to refer the papers cited in the following sub-sections:

5.1 Object Oriented Software Metrics

Object Oriented (OO) Software design has gained in importance over the past several years. Obviously software engineering folks are trying to be as efficient as possible while using this technique. But it so happens that sometimes they try to do so without proper backing in terms of theory or in terms of how their efficiency is measured. Existing metrics used in functional programming is bound to fail when applied to OO systems. The reasons are not hard to figure out. In this section we briefly discuss the important reasons why it is so and also some proposed methods in the literature. Of course it is not within the scope of this paper to go into too much detail; a separate paper like this one can be written just about object oriented metrics.

Some papers we found to be good starting point in understanding object oriented metrics and associated aspects are Purao et al [1], Bellin et al [10] and Dandashi [3]. We believe that there are a number of good – or even better – papers about object oriented metrics available in the literature, especially in the recent times. There are a lot of aspects which are unique in object oriented software design. These are summarized in no particular order of importance below:

1. The OO way of structuring software into classes, methods and such usually means, a metric like Lines of Code (LOC) is not applicable to it. But factors like number of methods, number of classes, number of messages passed between classes and methods – decides how tightly coupled the entities are, number of agent classes, depth of inheritance, breadth of a class, ratio of number of methods to classes, etc., could prove extremely useful. This is what Bellin et al propose in their paper [10] .

2. The paper mentioned in the previous point is quite basic and uses naive measures. The paper by Purao et al [1] is much more extensive. The authors of this paper have essentially surveyed a number of metrics applicable to object oriented software. They have suggested a framework which could be used in these systems. They also have a nice dimensional classification of OO metrics based on: direct and indirect metrics on the X-axis combined with elementary and composite metrics on the Y-axis. They go on to provide a formal framework to object oriented metrics, the description of which is out of the scope of this paper.

3. Dandashi [3] addresses a few related issues like applying existing (functional design) metrics to object-oriented systems and how the results correlate with expert prediction in certain cases and so on.

5.2 Metrics for Database application

In this section we quickly skim through a special case of using metrics. This is a proposal for use of a metric called "Database Points" (DBP) on database applications [7], more specifically on MS Access based ones. Extending it to

other database systems should be easily possible. Another thing which is worth mentioning is the fact that this metric is quite similar in semantics to the Function Point (FP) metric. This paper illustrates that in spite of differences in structural and functional aspects of database systems, applying metrics need not be done too differently.

The DBP metric is based on a typical organization of Access applications – usually consisting of five components: tables, relationships, transactions, forms and reports. Thus the authors propose five factors to their metric which they finally aggregate much the same way as function points are collected [8]. Each factor will be assigned to a difficulty level category – simple or average or complex (again this is a common method). Refer table 1 in the paper [7] for more details about this.

Going into a bit more detail, the authors classify each factor into sub-factors. For example, the table factor is sub-classified to sub factors like number of fields per table, properties per field, etc. As mentioned earlier, after each factor is assigned to a difficulty level category, a weighted average is computed, the result of which is the DBP metric score. Of course the authors have done tweaking of parameters sets and such to improve the effectiveness of their metric. There is also some empirical results shown in the paper.

5.3 Metrics for Software Evolution

Software evolution is another related area in which attempts to use metrics have been made. Mens et al provide a overview of such application of metrics in analyzing and improving software evolution in their paper [4]. They describe how it can be done in two ways – predictive and retrospective. Obviously predictive application of metrics to evolution of software is a bit more important since it can give insights about critical parts of the software even before it is developed.

In attempting predictive analysis of software evolution, the authors classify the software into different parts such as evolution-critical, evolution-prone and evolution-sensitive parts. The first term refers to parts of software that need to be evolved due to lack of quality – this might be due to a variety of reasons. The second term refers to parts that are likely to be evolved. The third term refers to parts which can lead to problems as it evolves. The authors describe these and also give details about how they can be used in their paper [4]. They also discuss retrospective analysis and suggest some future works in this area.

6 Constraints in Use of Metrics

In this section we look at factors which constrain the use of software metrics. These factors could be human, economic feasibility of applying metric programs and so forth. Here we concentrate on the human aspect and illustrate how it constrains the effectiveness of metric programs. Software designers and developers can lose interest in using metrics due to at least two reasons: one is the

degree of involvement of the concerned person, almost a purely psychological factor while the other is ease (or rather the difficulty) of metric use. We will look at these two factors in a bit more detail in the following sections – the former in section 6.1 and the latter in section 6.2.

6.1 Individual Involvement in Metric Development and Application

To illustrate the importance of individual involvement in developing and applying metrics to software development, we discuss a case study (paper) conducted by Slaughter [9] at Carnegie Mellon. In this paper, the author has researched and tried to explain why is it difficult to sustain a metrics program. According to empirical evidence only one in six metrics survive to see a second year of implementation. Is there a specific reason for this phenomenon? That is what she has researched on.

The author quotes some previous works and views that it is not the metrics themselves but rather resistance from management as well as developers which has lead to ineffectiveness of metrics – more of a cultural and psychological reason. Then she goes on to explain results from her case study. In the context of this paper, metrics are considered to be a feedback while the goal corresponds to the objective of the metrics. This could be considered a general notion. The author has worked on three hypotheses:

1. If the individual's perceived validity of and awareness about the metric is high, then he/she is bound to use it more (effectively).

2. If there is a high level of individual participation in setting the goals of a metric, then again the metric is bound to be used more. A couple of previous works on this is cited to emphasize the importance of managers having subordinates participate in setting project goals (which correspond to the metrics' goals).

3. Considering the above two hypotheses as a base, it is argued that effectiveness of metrics will ultimately depend on how frequently it is used. As the amount of feedback increases, it will be more effective.

After proposing the above hypothesis, the author explains the conceptual model based on the the same and also her methodology in analyzing a real time system. This includes a lot of manual paper work in which managers and developers had to do interviews, surveys and the like. Some of the main points that came out of the analysis are listed below. From both the developers' and maintenance personnel's perspective the main issues reported were:

- Lack of time to complete defect reports
- Negative consequences of reporting defects
- Not enough feedback from higher ups after the subordinates put in lot of time and effort turning in metric reports

- Not enough communication between developers and maintenance folks
- Function points are very subjective
- The participants did think that detecting and trying to prevent defects was a good thing to do
- Also it was agreed that function points can give useful insights in a lot of situations

The author at the end also gives some statistics with respect to ratio of folks who use metrics effectively in various levels – managerial, designer, developer, etc.

6.2 Ease of Metric Use and Overhead

Another important factor to consider when dealing with the human element is the ease or the difficulty of metric use. It has been proved time and again that if the overhead – collecting and analyzing data – of using metrics is really high then developers and managers stop using them effectively after a short period of time. Johnson et al discuss exactly this point in their paper [2]. They have delved into the reasons behind the failure of Carnegie Mellon (CMU)'s Personal Software Process (PSP) in terms of longevity of use.

The PSP project [6] was started with the objective of shifting the perspective of looking at metrics from an organizational level to an individual level. The system works by collecting size, time, defect data, etc from a project development team (on an individual basis) and then doing analysis on this data to improve project estimation and quality assurance. There were some basic assumptions – analysis of metric data can prove beneficial to any individual and that developers will continue to use those metrics for a long period of time. Johnson et al argue that PSP, in spite of the fact that it provided useful insights for software engineering students in quality assurance, is not used once the students quit the academic setting.

The authors of [2] then go on to discuss two of their own systems which facilitate, up to varying degrees, the collection and analysis of metric data. Leap was a system developed by these folks (references are available in the paper) and it automated the analysis part of the above stated problem. Yet they realized to their surprise that this system also failed to be used after a short period of time. This is argued to be due to the common requirement in both PSP and Leap, what the authors call "context switching" – the fact that the developers still need to switch between their primary task of developing software and their secondary task of collecting metric data.

To circumvent the problem with context switching, the authors have developed a new system called Hackystat [2] which automates both the metric data collection as well its analysis. They describe the architecture of their system in the paper. It consists of sensors associated with development tools which contacts a web-server whenever there is some newly available metric data. Thus

metric data is periodically collected and stored in an XML database and also analyzed automatically. If there is any odd or important phenomenon found in the analysis, the users are notified. Developers can also login and take a look at their performance and the like. This sounds like a reasonable way of dealing with the above stated problems.

The new system described in the previous paragraph does have its own drawbacks as the authors themselves admit. One of the main concerns has to do with privacy of the developer. As it is obvious, the system maintains detailed logs of every relevant step taken by the programmer, so developers might get apprehensive or irritated by the "big brother" notion the system brings in. Another important drawback of a system like Hackystat is its extreme programming language dependence. It will be very hard to generalize a system like this. Yet there are quite a few advantages too. The system does not suffer from problems like users forgetting or intentionally avoiding collection of metric data.

7 Conclusion

In this paper, we have attempted to give an introduction to software metrics and then go on to show improvements, validations performed on them, etc. Most of the content as has been indicated by various citations is adapted from various publications in the literature. It is clear that software metrics are here to stay and that the software engineering community is bound to use metrics a lot more in the coming years. Just to recap, metrics give us the ability to evaluate software process as well as the final software product. They are used in making quantitative/qualitative decisions as well as in risk assessment and reduction. We have tried to show why validating metrics is crucial in producing effective ones. Some new improved methods have been explained briefly. Of course a paper like this would not have been complete without mention of the effect of the human element – on the use of metrics in this case. We have attempted to explain some issues in this aspect as well. In essence this paper can be considered a starting point to learn about software metrics since it contains information from a variety of sources.

References

[1] Sandeep Purao and Vijay Vaishnavi, Product metrics for object-oriented systems, ACM Computing Surveys (CSUR), Volume 35, Issue 2, pp 191-222, 2003

[2] Philip M. Johnson, Hongbing Kou, Joy Agustin, Christopher Chan, Carleton Moore, Jitender Miglani, Shenyan Zhen and William E. J. Doane, Beyond the Personal Software Process: metrics collection and analysis for the differently disciplined, Proceedings of the 25th international conference on Software engineering, Portland, Oregon, pp 641-646, 2003

[3] Fatma Dandashi, A method for assessing the reusability of object-oriented code using a validated set of automated measurements, Proceedings of the 2002 ACM symposium on Applied computing, Madrid, Spain, pp 997-1003, 2002

[4] Tom Mens and Serge Demeyer, Future trends in software evolution metrics, International Conference on Software Engineering, Proceedings of the 4th international workshop on Principles of software evolution, Vienna, Austria, pp 83-86, 2001

[5] Norman E. Fenton and Martin Neil, Software Metrics: Roadmap, International Conference on Software Engineering, Limerick, Ireland, pp 357-370, 2000

[6] Watts S. Humphrey, The Personal Software Process (PSP), CMU-SEI-2000-TR022, 2000

[7] Sana Abiad, Ramzi A. Haraty and Nashat Mansour, Software metrics for small database applications, ACM Symposium on Applied Computing, Como, Italy, pp 866-870, 2000

[8] Roger S. Pressman, Software Engineering: A Practitioner's Approach, McGraw-Hill, 1996

[9] Sandra Slaughter, Assessing the use and effectiveness of metrics in information systems: a case study, Proceedings of the 1996 ACM SIGCPR/SIGMIS conference on Computer personnel research, pp 384-391, 1996

[10] David Bellin, Manish Tyagi and Maurice Tyler, Object-oriented metrics: an overview, Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research, Toronto, Ontario, Canada, 1994

[11] Lem O. Ejiogu, A unified theory of software metrics, Proceedings of the 1988 ACM sixteenth annual conference on Computer science, pp 232-238, 1988

[12] Dennis Kafura, A survey of software metrics, Proceedings of the 1985 ACM annual conference on The range of computing : mid-80's perspective, Denver, Colorado, pp 502-506, 1985

[13] Dennis Kafura and James Canning, A validation of software metrics using many metrics and two resources, International Conference on Software Engineering, London, England, pp 378 - 385, 1985

[14] J. E. Gaffney, Metrics in software quality assurance, Proceedings of the ACM CSC-ER '81 conference, pp 126-130, 1981