SOFTWARE METRIC ANALYSIS METHODS FOR 
PRODUCT DEVELOPMENT / MAINTENANCE PROJECTS

Mr. S. Manivannan  
Research Scholar  
Anna University Coimbatore  
Email: smanivannan2003@yahoo.com

Dr. S. Balasubramanian  
Research Supervisor, Anna University Coimbatore  
E-Mail: s_balasubramanian@rediffmail.com

ABSTRACT

This paper presents an overview of the collection, analysis, and reporting of software metrics. Only the progress, effort and trouble report metrics are required for the project. However, the student should be familiar with all the metrics described below.

Software metrics are numerical data related to software development. Metrics strongly support software project management activities. They relate to the four functions of management as follows:

1. Planning - Metrics serve as a basis of cost estimating, training planning, resource planning, scheduling, and budgeting.
2. Organizing - Size and schedule metrics influence a project’s organization.
3. Controlling - Metrics are used to status and track software development activities for compliance to plans.
4. Improving - Metrics are used as a tool for process improvement and to identify where improvement efforts should be concentrated and measure the effects of process improvement efforts.

A metric quantifies a characteristic of a process or product. Metrics can be directly observable quantities or can be derived from one or more directly observable quantities. Examples of raw metrics include the number of source lines of code, number of documentation pages, number of staff-hours, number of tests, number of requirements,
etc. Examples of derived metrics include source lines of code per staff-hour, defects per thousand lines of code, or a cost performance index.

INTRODUCTION ON SOFTWARE METRICS

Software development process and projects have a long and storied history of failure. In fact, 82% of projects today run late, while errors cost 80% of the average project budget to fix (The Standish Group). Certainly no other business process today is allowed to endure this sort of failure. But software development is often left to chance, despite the significant cost and importance of the process.

Why are software projects so prone to failure? In large part, it’s because of a profound lack of visibility and transparency into development processes. And this lack of visibility only increases with the complexity of projects and physical distribution of teams. This means that projects that are sent offshore are even more challenged when it comes to visibility, transparency and control.

Managers who believe they understand metrics programs are advised to take another look.

Today’s approach to metrics calls for automation over process and in-process adjustment over post-mortem analysis. And instead of a centralized approach to management control, today’s metrics programs are inclusive, making information available to all stakeholders in the lifecycle. And rather than trying to serve up a sea of data just because it’s available, today’s programs are based on manageable scale and scope and are designed to deliver simple, actionable insights.

Of course, no metrics program is a silver-bullet solution. To be effective, metrics must be properly planned, managed and acted upon. What is measured, how it’s collected and how it’s interpreted are the difference between brilliant insights and blind alleys on the path to metrics-driven software development. This whitepaper discusses the five essential elements of software development metrics success, providing a framework for planning a metrics program within your organization.

SOFTWARE DEV PHASES AND METRICS

A software development process is a structure imposed on the development of a software product. Synonyms include software life cycle and software process. There are
several models for such processes, each describing approaches to a variety of tasks or activities that take place during the process.

**SOFTWARE DEVELOPMENT ACTIVITIES**

The activities of the software development process represented in the waterfall model. There are several other models to represent this process.

**PLANNING**

The important task in creating a software product is extracting the requirements or requirements analysis. Customers typically have an abstract idea of what they want as an end result, but not what software should do. Incomplete, ambiguous, or even contradictory requirements are recognized by skilled and experienced software engineers at this point. Frequently demonstrating live code may help reduce the risk that the requirements are incorrect.

Once the general requirements are gleaned from the client, an analysis of the scope of the development should be determined and clearly stated. This is often called a scope document.
Certain functionality may be out of scope of the project as a function of cost or as a result of unclear requirements at the start of development. If the development is done externally, this document can be considered a legal document so that if there are ever disputes, any ambiguity of what was promised to the client can be clarified.

**DESIGN**

Domain Analysis is often the first step in attempting to design a new piece of software, whether it be an addition to an existing software, a new application, a new subsystem or a whole new system. Assuming that the developers (including the analysts) are not sufficiently knowledgeable in the subject area of the new software, the first task is to investigate the so called "domain" of the software. The more knowledgeable they are about the domain already, the less work required. Another objective of this work is to make the analysts, who will later try to elicit and gather the requirements from the area experts, speak with them in the domain's own terminology, facilitating a better understanding of what is being said by these experts. If the analyst does not use the proper terminology it is likely that they will not be taken seriously, thus this phase is an important prelude to extracting and gathering the requirements. If an analyst hasn't done the appropriate work confusion may ensue: "I know you believe you understood what you think I said, but I am not sure you realize what you heard is not what I meant."[1]

**ARCHITECTURE**

The architecture of a software system or software architecture refers to an abstract representation of that system. Architecture is concerned with making sure the software system will meet the requirements of the product, as well as ensuring that future requirements can be addressed. The architecture step also addresses interfaces between the software system and other software products, as well as the underlying hardware or the host operating system.

**IMPLEMENTATION, TESTING AND DOCUMENTING**

Implementation is the part of the process where software engineers actually program the code for the project.

Software testing is an integral and important part of the software development process. This part of the process ensures that bugs are recognized as early as possible.
Documenting the internal design of software for the purpose of future maintenance and enhancement is done throughout development. This may also include the authoring of an API, be it external or internal.

**DEPLOYMENT AND MAINTENANCE**

Deployment starts after the code is appropriately tested, is approved for release and sold or otherwise distributed into a production environment.

Software Training and Support is important because a large percentage of software projects fail because the developers fail to realize that it doesn't matter how much time and planning a development team puts into creating software if nobody in an organization ends up using it. People are often resistant to change and avoid venturing into an unfamiliar area, so as a part of the deployment phase, it is very important to have training classes for new clients of your software.

Maintenance and enhancing software to cope with newly discovered problems or new requirements can take far more time than the initial development of the software. It may be necessary to add code that does not fit the original design to correct an unforeseen problem or it may be that a customer is requesting more functionality and code can be added to accommodate their requests. It is during this phase that customer calls come in and you see whether your testing was extensive enough to uncover the problems before customers do. If the labor cost of the maintenance phase exceeds 25% of the prior-phases' labor cost, then it is likely that the overall quality, of at least one prior phase, is poor. In that case, management should consider the option of rebuilding the system (or portions) before maintenance cost is out of control.

Bug Tracking System tools are often deployed at this stage of the process to allow development teams to interface with customer/field teams testing the software to identify any real or perceived issues. These software tools, both open source and commercially licensed, provide a customizable process to acquire, review, acknowledge, and respond to reported issues.
<table>
<thead>
<tr>
<th>Measures</th>
<th>Formula</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Effort</td>
<td>In person months/man months</td>
<td>Estimation sheet/MPP/Schedule</td>
</tr>
<tr>
<td>Variance</td>
<td>Actual effort-estimated effort/estimated effort * 100</td>
<td>MPP/Schedule</td>
</tr>
<tr>
<td>Productivity</td>
<td>Estimate effort*100/Actual effort</td>
<td>MPP/Schedule</td>
</tr>
<tr>
<td>Schedule variance</td>
<td>Actual end date-Planned end dated</td>
<td>MPP/Schedule</td>
</tr>
<tr>
<td>Review effort</td>
<td>Time spent on reviews</td>
<td>Technical Review Report’s</td>
</tr>
<tr>
<td>Rework effort</td>
<td>Time spent on rework</td>
<td>Activity sheet</td>
</tr>
<tr>
<td>Requirement changes</td>
<td>Changes raised after design phase</td>
<td>Change Requests</td>
</tr>
<tr>
<td>Unscheduled effort</td>
<td>Other activities in the project</td>
<td>Activity Sheets/MPP</td>
</tr>
<tr>
<td>Risk Analysis</td>
<td>Occurrence/Impact</td>
<td>Risk Management Plan</td>
</tr>
<tr>
<td>Phase defect containment efficiency</td>
<td>FORMULA ( \frac{A}{A+B} )</td>
<td>Technical Review Report’s/Defects Report</td>
</tr>
<tr>
<td></td>
<td>( A = ) No of problems detected within that phase through reviews</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( B = ) No of problems attributable to that phase (during defect analysis)</td>
<td></td>
</tr>
<tr>
<td>Defect removal efficiency</td>
<td>( \frac{A}{B} \times 100 )</td>
<td>Technical Review Report’s/Defects Report/Support ease system</td>
</tr>
<tr>
<td>Test effectiveness ratio</td>
<td>No of test defects / Total no of defects</td>
<td>Technical Review Report’s/Defects Report</td>
</tr>
<tr>
<td>Defect classification</td>
<td>Root cause of the defect using fish bone.</td>
<td>Causal analysis report</td>
</tr>
</tbody>
</table>
METRICS ANALYSIS METHODS

Pie Chart

The pie chart is the simplest type of graph. It is used to show the distribution of up to about ten data points at one moment in time.

Pareto Diagram

Pareto diagram helps to focus or prioritize the issues that will have the greatest impact if solved. Based on the proven Pareto principle as 20% of the sources causes 80% of any problem or issue. It displays relative importance of problems or issues in a simple, quickly interpreted, visual format.

Guidelines for usage:

- Identify the issue or problem to be monitored
- Choose the time frame for study
- Collect data (no of occurrences or frequency) on each problem category
- Calculate relative frequency or % of occurrence of each problem category
- Sort problem category in descending order of relative frequency.

Draw pareto diagram by making problem category on horizontal line [X axis] with the bars above each problem category from left to right to indicate relative frequency or % on the vertical line [Y-axis]

A Pareto Diagram is a special type of bar chart used to identify problem areas what do we need to fix first? What are the biggest fires to put out?
It is useful in a software environment because defects tend to cluster in portions of code, so if we can identify where a lot of defects have already been found, we can probably find more defects in those same areas.

Here we would typically find the defect rate for all major software components, and then plot the defect rate (Y axis) for each component (X axis). The Pareto diagram lists the components in descending order of defect rate, which automatically puts the most defect ridden components on the left of the graph.

There is an optional line on a Pareto diagram, which shows the cumulative percent of all cases for each component, going from left to right. Hence the line always reaches 100% by the right-most X value, and strictly increases going from left to right. In the context of defect rate analysis, this line would not have much meaning since defect rates cannot be added.

![Pareto Diagram of Component Defect Rate](image)

**Column Bar Chart**

There are many variations possible on the histogram.

- The bars can be oriented vertically (a.k.a. a column chart), or horizontally. Usually vertical bars are preferred.
- A histogram can contain one measure (a simple bar chart) or more than one. The latter is a stacked bar chart if the bars are placed on top of each other, or a clustered bar chart if the bars are sitting next to each other.
- The bars can represent the actual value of each measure (e.g. the number of problem reports in various statuses), or they can show the percent each measure
contributes to the total (e.g. the percent of problem reports which are in each status).

- A third possibility, for real-valued data (as opposed to integer data), is that each bar can represent “the number of data points within a specific range of values.” So a histogram could show the distribution of problem report closure times, where one bar might be the “the number of problem reports which closed in 2 or 3 days,” and another bar might be the “the number of problem reports which closed in 1 day or less,” and so on.
LINE CHART

A run chart, by definition, graphs one or measures over time. Hence the X axis for a run chart is always some form of measure. By using straight lines to connect data points for each measure, the line chart allows more data to be presented than could easily fit on a histogram.

Scatter Diagram

Scatter diagrams are used to graph two measures against each other. They are generally used to determine if there is a correlation between the two measures, e.g. “does testing time decrease when more time is spent doing review?”
In a scatter diagram the Y axis is the measure we wish to predict or understand more fully, and the X axis is the measure which affects the value of the Y axis. So in the example, the testing time would be the Y axis, and review time would be the X axis.

A scatter diagram can be used to plot data points initially. Then, based on the type of relationship which appears to exist between the measures, a regression analysis is done to determine if that relationship is statistically meaningful.

![Current Salary vs. Educational Level](image)

**Radar Diagram**

Radar diagram is a structure conceptual space along vectors or axes representing specified variables where one can map where any given object of study falls on each vector or axis, and produce a composite profile by linking the points on each axis to create a polygon.

To draw the radar diagram, the values must first be articulated. Then, one must determine whether the values are unipolar or bipolar:

- **Unipolar** – lack an opposite; expressed as a vector (from origin out in one direction).
- **Bipolar** – imply an opposite end; expressed as an axis or spectrum (values extending out on either side of the origin).

Radar diagrams, or similar visual approaches to self-anchoring evaluative scales – may assist in analyzing and comparing value structures in either scenarios of possible futures, or visions articulating preferred futures. Radar diagrams may also assist in the
conscious design of value structures both for scenarios of possible futures, and for visions of preferred futures. In visioning, radar diagrams are a useful tool first for organizing and clustering values, and second for comparing values among the members of the community engaged in the visioning process.

### Control Chart

The control chart is a special kind of line graph used to monitor performance of a measure. The main purpose of using a control chart is to provide statistically sound basis for determining if a measure is under control.

The main feature of a control chart is that there are three lines in addition to the measure itself. These three are the mean, the Upper Control Limit (UCL) and Lower Control Limit (LCL). There are sets of rules for determining if your measure is “out of control” based on its behavior relative to the mean, UCL and LCL.

There are many kinds of control charts to meet a wide range of measure types.

- **Variable Data control charts** apply when the measure is a continuous (real-valued) quantity, such as temperature, pressure, cost, etc. The control charts for variable data are based on the mean value of the measure and some measure of its standard error.
• Attribute Data control charts apply when the measure is a discrete event, such as a defective part, an absent employee, or some other measure based on integer variables.

Of those, software tends to focus on defect-oriented measures, so Attribute Data control charts apply most often here. Within Attribute Data control charts, there are different methods based on whether each defect is counted separately (called Defect data), or whether entire units are passed or failed (Defective data). Then the type of control chart is selected based on whether the samples are constant size (rare in software), or variable size. Assuming the latter applies, the choice of control chart becomes this:

• If each defect is counted separately (Defect data), and the sample size may vary, use the ‘u’ type of control chart. This is the ‘number of defects per unit’ type of control chart.

• If each unit is passed or failed as a whole (Defective data), and the sample size may vary, use the ‘p’ type of control chart. This is the ‘fraction defective’ type of control chart.

After selecting the type of control chart, there are formulas for calculating the UCL and LCL. Hence every data point for the measure also results in calculation of the control limits. (If the sample sizes vary by less than +/- 20%, then the control limits can be fixed, which makes the chart easier to read.)
CAUSE AND EFFECT DIAGRAM (FISHBONE)

A cause and effect diagram isn’t a method for presenting measures graphically. It is a tool for analysis of the causes of some undesired effect (problem). It’s called a fishbone diagram because it resembles the skeleton of a fish.

Cause and effect diagrams are generally generated in a meeting setting, by brainstorming about some problem. The diagram is a way of capturing everyone’s thoughts.

The ‘head’ of the diagram is a box containing the effect you wish to cure. A horizontal line from the head forms the backbone of the ‘fish.’ The lines at 45 degrees to the backbone are the major types of causes of the effect. Some common choices for major causes are:

- Machines (Plant, or equipment)
- Methods (or Processes)
- Materials (raw materials for the product)
- People (staffing)
- Policies (high level rules)
- Procedures (steps in a task)
- Environment (facilities)
- Measurement (data collection)

The choices for major causes need to be customized for each problem. Generally at least four major causes are selected.

Specific possible causes of the effect are shown by horizontal lines which point to each major cause. Each possible cause can be elaborated upon by adding more lines to show the causes of the possible cause. This process can continue to any desired level of detail.

The main questions being asked to generate the cause and effect diagram are:

- Why does that happen?
- What causes that?
- What influences that?
The fishbone diagram can be used to support causal (root cause) analysis.

LIMITATIONS

It is very difficult to satisfactorily define or measure "how much" software there is in a program, especially when making such a prediction prior to the detail design. The practical utility of software metrics has thus been limited to narrow domains where they include:

• Schedule
• Size/Complexity
• Cost
• Quality

Too much emphasis on any one of these aspects of performance is likely to create an imbalance in the team’s motivations, leading to a dysfunctional project.

REFERENCES


2. Dr. Cem Kaner, Software Engineer Metrics: What do they measure and how do we know?

