McCabe SOFTWARE

White Paper

Baseline Code Analysis Using McCabe IQ

Software managers are always looking for better and innovative ways to make software development more efficient and effective throughout the product life cycle. Development and quality assurance (QA) managers often ask questions such as: How can I get the code into production faster? What code should I refactor? How should I best assign my limited resources to different projects? How do I know if code is getting better or worse as time goes on?

Such questions are especially difficult to answer in today's business environment. Answering them with any degree of confidence presumes intimate knowledge of the code being managed, something that is becoming less and less feasible for managers without the help of additional tools. For example, software is growing to previously unimaginable levels of size and complexity, and it is more and more common for the code being managed to have been developed by someone else, such as a third party vendor or a different department within the company. In either case, the code resists comprehensibility: What are its components? How are they all related? Which are the critical ones for maintenance, enhancement, and testing initiatives?

DeMarco's insight in 1982 has now become a well-known adage in software engineering: Managers cannot control or manage what they cannot measure. What managers need are measurements that makes the source code comprehensible in terms of structure (showing how all the different modules fit together) and quality (revealing the relative size, complexity, and testedness of the various modules). Baseline code analysis, the subject of this paper, is a method of establishing such measurements in order to help managers make more informed decisions about the product throughout its life cycle.

Purpose of this Paper

This document has been written to provide the answer to three basic questions:

- What is baseline code analysis and why is it important?
- What are the challenges of baseline code analysis?
- How can baseline code analysis with McCabe IQ be used to add value to Development and QA processes?

Table of Contents

What is Baseline Code Analysis? 2

Importance of Baseline Code Analysis.................................2

The Objectives of Baseline Code Analysis.................................4

Best Practices for Baseline Code Analysis.................................4

Challenges of Baseline Code Analysis.................................7

McCabe IQ Functionality for Baseline Code Analysis 8

The McCabe IQ Advantage.........10

Appendix A: McCabe IQ Metrics By Category.......................11

Who Should Read This Paper?

Software managers who are:

- In need of accurate, profitable representations of the code they manage
- Establishing or restructuring their company's QA department and who want to equip it with state of the art software management technologies.

T: 800-638-6316
www.mccabe.com

McCabe SOFTWARE

What is Baseline Code Analysis?

Any kind of comprehensive analysis of source code can be considered baseline code analysis. In essence, baseline code analysis identifies and quantifies the elemental building blocks of a given program, for example, its various lines of code, modules, and logical paths. With this data, baseline code analysis can provide structural and metrical representations of the code.

Structural Representations

Baseline code analysis can provide a 'map' of the program's components, from the highest-level view of all modules and their calling hierarchy to the lowest-level view of an individual path (decision sequence) within a module. As programs grow in size and complexity, structural representations make them instantly comprehensible and therefore more manageable.

Metrical Representations

Baseline code analysis can provide software metrics - quantitative data on various measurable elements of source code. Metrics are either collected (by measuring such things as lines of code, the number of logical paths, and the number of 'children') or calculated (derived from measurements using mathematical formulas).

Meaningful Thresholds

Numerous independent studies as well as standards published by the National Institute of Standards and Technology (NIST) have established threshold guidelines that turn source code metrics into reliable indicators of software quality

Metrics are generally collected and calculated on a module-by-module basis, they are used to highlight areas of code that exhibit significantly low or high values vis-à-vis established threshold values. For example, independent studies indicate that the likelihood of introducing a defect into a module you are modifying increases exponentially if that module has more than 10 logical paths. Baseline code analysis can highlight the modules that have more than 10 logical paths, allowing managers to flag them as particularly risky modules to modify. Other characteristics that metrics can reveal on a module-by-module basis are maintainability, reusability, redundancy, complexity, and testedness.

Importance of Baseline Code Analysis

The primary importance of baseline code analysis is that it helps managers to understand, in software engineering terms, the code they have to manage. Further, baseline code analysis provides objective representations of the structure and quality of the code being managed. This means that the code

McCabe SOFTWARE

itself is the final reference point for management decisions, taking guesswork, intuition, and other unreliable methodologies out of the picture. Baseline code analysis is also important for the following reasons.

Increased Effectiveness of Development and QA Efforts

Baseline code analysis objectively identifies the most complex areas of the code - that 20% of the code that accounts for 80% of the problems. With this knowledge, managers can allocate appropriate resources. They can be assured that their most experienced developers are assigned to the most critical areas of the program; they can focus testing resources where modifications to a program are most likely to have introduced defects.

The ability to pinpoint problematic code early in the production cycle is also an important feature of baseline code analysis. Studies have shown that the cost of finding and fixing a defect increases as software is promoted through its life cycle. By targeting problematic code from the beginning of its life cycle, baseline code analysis encourages early defect detection and prevention. In short, baseline code analysis allows software managers to allocate the most cost-effective resources to the most critical parts of the code at the most advantageous points in the product cycle.

The 80-20 Rule

It is well known that on average 20% of the code accounts for 80% of the resource usage, problems, and errors - i.e., the sum of expenditures - spent throughout the product cycle. That 20% is comprised precisely of the most complex segments of the code, and baseline code analysis is the best tool available for isolating it. With baseline code analysis, that critical 20% can be recognized from the beginning of the product cycle and taken into account in all decisions throughout the QA process.

Increased Efficiency of Development and QA Effort

Baseline code analysis provides managers with crucial information for streamlining development and QA efforts. For example, it allows QA managers to instantly identify untested areas of a program under test. With this knowledge, managers can streamline test plans, short-cutting the quest for meaningful tests and preventing such common problems as overtesting (spending excessive time on areas of the program that are at low risk for defects) and redundant testing (testing the same functions over and over again).

Historical Trend Analysis

As the program evolves from revision to revision, it changes in terms of size and quality. Baseline code analysis provides a historical record of each revision's metric values so that managers can be sure the trends are not moving in undesirable directions - for example, getting unnecessarily complex and difficult to test, or getting more and more unstructured and costly to maintain (the hallmark of "spaghetti code").

More on Trend Analysisis

For a more detailed discussion of McCabe IQ and trend analysis, read the McCabe paper titled, "McCabe Recommended Approach to Code Reviews."

McCabe SOFTWARE

A Typical Scenario

In today's market, software is often produced through third party arrangements, or purchased from other companies. Baseline code analysis provides valuable insight into the code that is being transferred.

Maintenance departments that often take the ownership of the developed code can perform baseline code analyses to become familiar with the code characteristics of inherited programs in order to allocate appropriate resources for future enhancements and bug fixing.

The Objectives of Baseline Code Analysis

Given the structural representations and the range of metrics it provides, the fundamental objectives of baseline code analysis can be summarized as follows:

- To objectively identify the most complex and unstructured parts of the code. Baseline code analysis can be used to highlight the parts of the code that exceed given metrics thresholds. Managers get answers to such questions as: Where is the code going to be particularly difficult to maintain? Where are the segments of the code most at risk for introducing defects? How has a change that has been made impacted the quality of the module?
- To evaluate the scope of software modifications.

  Structural analysis can reveal all modules of a program that are related in the calling hierarchy to a module that is targeted for modifications. Managers get answers to such questions as: How widespread is a proposed change's potential influence? What parts of the code need regression testing (and which parts do not need regression testing!) as the result of a change?
- To assist managers in making informed decisions about allocating resources for testing.

  Baseline code analysis can identify a collection of testable areas of the program, and it can determine which areas have and have not been covered by a given set of tests. Managers get answers to such questions as: How much testing should be performed? How much testing is left to be done? On which parts of the code should testing be focused?
- To facilitate software refactoring/reengineering efforts.

  Metrics can reveal areas of the code that are good candidates for refactoring - that is, areas that, if refactored, would mean substantial savings in time and resources down the road. Managers get answers to such questions as: Which parts of the software are redundant or reusable? Which parts are costly to maintain, or overly complex ('spaghetti' code)?

Best Practices for Baseline Code Analysis

This section outlines practices shared by companies that have profitably incorporated baseline code analysis into their development and QA processes. The most successful baseline code analysis enterprises tend to:

- Develop a measurement program
- Streamline and continuously improve Development and QA processes
- Automate!

McCabe SOFTWARE

Best Practice #1

Develop a Measurement Program

There are two primary imperatives for establishing a measurement program:

- To identify metrics of interest for your development environment
- To establish a set of policies for the regular generation of reports.

Identify Metrics of Interest

It is crucial to identify a set of metrics that can be used compositely to represent the code in certain ways in order to address specific management goals. In developing a set of metrics of interest, focus on four major characteristics of the software: size, quality, risk, and readiness.

McCabe IQ Metrics

For a list of specific metrics returned by McCabe IQ in each category - size, quality, risk, and readiness - refer to Appendix A at the end of this document.

- **Size.** Use a set of metrics that represent the size of the program. Software size can be measured with a variety of metrics (for example, line and branch counts) and is a good initial indicator of code complexity.
- **Quality.** 'Good quality' code is code that is well structured and therefore easy to understand, modularize, and maintain. Use a set of metrics that provide a measure of unstructured code segments. For example, an excessive number of class couplings can be an indicator of unstructuredness.
- **Risk.** Use a set of metrics that indicate the likelihood of finding or introducing defects in the various code segments. Advanced indicators of code complexity are essential here. Whenever code gets exceedingly complex - for example, deeper in the inheritance hierarchy, or less cohesive due to the dissimilarity between methods in a class - it can be considered at greater risk for defects.
- **Readiness.** Use a set of metrics that measure tested segments of the code. Metrics can be acquired, for example, that indicate tested lines, branches, and paths. Metrics that measure tested segments of the code can be used to determine the 'readiness' of branches and paths - that is, the extent to which they can be considered stable and defect-free.

Establish Policies for Continuous Reporting

Once you have isolated a set of metrics that address your specific management needs, it is important to consider baseline code analysis' place within the Development and QA process. At what intervals should reports be generated to monitor software metrics? Or, what events should trigger the generation of what reports? Ideally, baseline code analysis is not only used to provide a one-time insight into the code of interest but is also used regularly to provide a means for monitoring the code at intervals throughout the development cycle.

McCabe SOFTWARE

Another Best Practice:
Take Control of the Software Development Process

Once problematic code is identified by baseline code analysis, management is faced with a welcome challenge - cost-effective choices in the interests of a reliable and stable product. For example, should appropriate resources be allocated to accommodate it? Or should the code be targeted for refactoring? Choosing the former can often be a time saver up front, but it trades-off on maintainability down the road. On the other hand, depending on the development environment, choosing the latter may be both ideal and feasible.

Best Practice #2

Streamline & Continuously Improve Development and QA Processes

The information baseline code analysis provides can be used to make software development, maintenance, and testing efforts more efficient and effective. Some proven strategies for streamlining and improving these areas of production are discussed below.[1]

Improve Software Change Initiatives

- Make decisions about scheduling based on an assessment of the scope of proposed changes.
- Make decisions about resource allocations based on the complexity of modules targeted for change. Assign more experienced resources and more time to design, change, and review the complex code.
- Establish a change/version control process. As the program changes in development, perform code analyses at certain milestones, and compare the results with the baseline code analysis to evaluate the trend in code complexity and other aspects of the software.

Improve Testing

- Establish a threshold for the completion of testing using the determination baseline code analysis makes of testable units of code.
- Use metrics that indicate tested paths and branches of code to streamline test plans and track the degree of 'readiness' of the code.
- Use risk and complexity analyses to establish guidelines for how much testing should be performed for different components of the software. Perform more rigorous path coverage testing for the parts of the program most at risk from defects.

Streamline Software Redevelopment Efforts

- Use customized reports and code visualization to identify the “outliers” - code segments that register metrics values outside the normal range for the software. Outliers identify code that is hard-to-maintain and error-prone.

---

[1] McCabe provides white papers specifically addressing the benefits of baseline code analysis for the three Development and QA processes discussed in this section (software change initiatives, testing/QA, and reengineering). For a more in-depth treatment of how baseline code analysis can help in these areas, refer to the appropriate white paper: “McCabe Recommended Approach to Software Change Analysis,” “Improved Testing Using McCabe IQ Coverage Analysis,” or “McCabe Recommended Approach to Code Reviews.”

- Focus redevelopment efforts on the outliers, particularly when they are functionally critical. Simplify and restructure such code segments to make them less costly and more stable.
- Use detailed architectural analysis to pinpoint redundant code. Target redundant code to reduce the system's overall complexity and size, easing overall maintenance requirements and increasing the performance.

Best Practice #3

Automate!

Use automated tools to collect metrics and to generate reports. And use automated tools that are flexible and can be adapted to your analysis needs - that is, tools that do not force you to adapt your analysis environment to their functionality.

Challenges of Baseline Code Analysis

Baseline code analysis can be difficult to implement for many reasons. To begin with, management is faced with the need of selecting useful metrics and establishing meaningful thresholds. What, for example, is an objective and useful definition for code complexity? A meaningful answer to this question will reflect a great deal of research into which metrics can be used as indicators of what aspects of code quality.

Once a dependable set of metrics and thresholds is acquired, how can the cavalcade of numeric data be presented such that it gives a comprehensive picture of the code under analysis? Visualizing the results of baseline code analyses is another high-order challenge, because a greater amount of information can cause more problems than it solves if it cannot be channeled into understandable and meaningful representations.

In other ways companies are challenged with making the most out of the information baseline code analysis provides. Every stage of the product life cycle can benefit from greater knowledge of the code in production, from planning to development to testing. These efforts are often undertaken by different managers, all devising different applications for the same fundamental set of metrics. Integrating baseline code analysis across various Development and QA groups cohesively and uniformly is important because it insures the greatest communicability between stages in the development process. Achieving cohesiveness and uniformity, however, can be difficult.

McCabe SOFTWARE

McCabe IQ Functionality for Baseline Code Analysis

McCabe IQ is a tool that can be used to perform baseline code analysis. Its key functionality for the purposes of baseline code analysis includes:

- Metrics
- Reports
- Visualization
- Automation
- Integration

McCabe Metrics

Custom Metrics and Reports

If a user requires metrics not provided by McCabe IQ, he or she can import them from other systems.
McCabe IQ also allows users to define custom formulae in order to acquire special-needs derived metric values. Custom measurements and derived metrics can be included in the baseline code analysis to highlight aspects of the code not otherwise 'counted" by McCabe IQ. Similarly, custom reports can be designed to show specific ranges of metrics values, to factor in custom metrics, to sort data according to user specifications, and so on.

McCabe IQ can record up to 105 metrics for a given program. These metrics are gathered at the module (function/method/Perform Range) level, Program Level (one or more source files), or System Level (one or more programs). Some standard metrics that McCabe IQ returns are described in Appendix A of this paper.[2]

All metrics returned by McCabe IQ come with default threshold values, that is, the high and low values of what constitutes a "normal' or "ideal' range for the given measurement. System-supplied thresholds represent industry standards, but they can be adjusted, as needed.

McCabe Reports

McCabe IQ provides about 35 predefined reports from which users can choose. McCabe IQ reports range from simple spreadsheets to graphical representations of data that can be granulated for local (modular) and system analyses. To enhance the comprehensibility of the results of the baseline code analysis, all reports have hypertext capabilities that allow users to move easily between related data sets and system components.

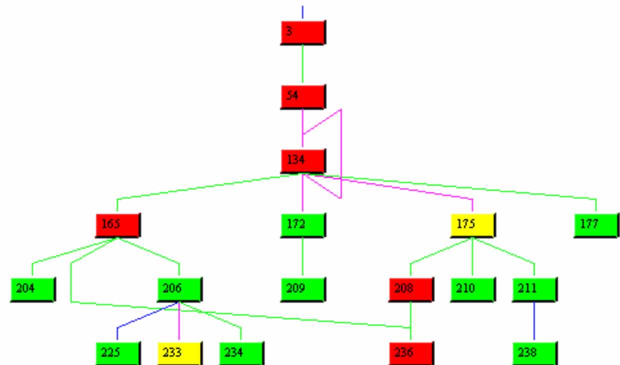
[2] Appendix A does not describe all of McCabe IQ's metrics. For more information, refer to the McCabe paper entitled, "Metrics and Thresholds in McCabe IQ."

McCabe Visualization

McCabe IQ provides extensive visualization capabilities at all levels of analysis. The basic McCabe IQ visualization tools are battlemaps and flowgraphs.

Battlemaps

Battlemaps display a structure chart that graphically represents the functional hierarchy of the program. The "exclude" tool of the Battlemap gives you the ability to focus on specific segments of the program. Battlemap uses c code complexity on the structure chart, allowing managers to immediately visualize the complex and critical parts of the code.



Battlemap. The rectangles represent modules; each module is color-coded to indicate whether its value for the selected metric is above (red), below (yellow), or within (green) the threshold.

Flowgraphs

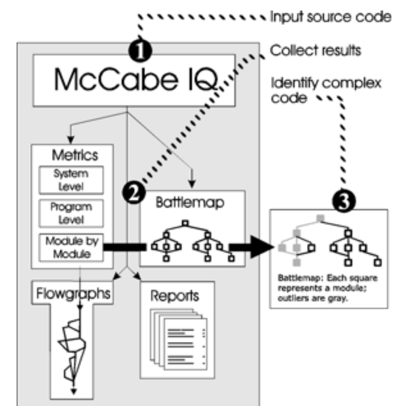
Flowgraphs graphically display the control structure of the code and path of execution inside each module (method, function, or subroutine.) Flowgraphs facilitate code comprehension and are used quite often for code reviews. They highlight unstructured behavior of the code, and the user can graphically see the cause of increased code complexity.

Each flowgraph is accompanied by an Annotated Source Listing (ASL) that provides a mapping between the flowgraph and the lines of code. The ASL allows users to view the specific lines of code that tend to contribute to the increased complexity of the code.

Automation

McCabe metrics and graphical displays can be obtained with the click of a mouse. You simply tell McCabe IQ the source code that you want it to analyze, and it

- 'Reads' the source code using powerful parsers
- Identifies and measures the program's various components
- Renders metrics and visuals.



Other McCabe IQ Functionality

- Class Editor

Interface for viewing, modifying, and creating classes. With Battlemap, allows users to group modules based on user-defined criteria in order to analyze the relationship among them.

- Data Dictionary

Tool for tracking data use in programs. It provides a view of local data, global data, parameters, and module declarations. Users can analyze complexity of code with respect to specific data usage.

- Metric Snapshots

Takes a picture of program metrics at various times in the development cycle and saves it to a repository - a critical tool for trend graphing.

- Compare

Tool for identifying functionally similar modules. Useful for identifying modules in which a found error might be repeated. Useful for identifying redundant code and dead code.

Integration

McCabe IQ is a fully integrated suite of code analysis products, each designed for addressing the specific needs of different stages in the product cycle. There are many tools on the market that can help you with specific software development tasks. But McCabe IQ's integrated suite of products is designed to manage your full software development needs within a single framework. From planning to development to testing, McCabe IQ brings simplicity, cohesiveness, and uniformity. "IQ" means "Integrated Quality."

The McCabe IQ Advantage

McCabe IQ is highly flexible, customizable, and configurable to most development environments and management needs. It fits itself to the manager's real-world analysis requirements, rather than fitting the analysis to its functionality.

McCabe IQ's source code analysis is automated, and platform and language independent. Source code from any platform can be brought to the platform on which McCabe IQ is installed for analysis. Additionally, McCabe IQ can analyze most programming languages and "dialects". The complete list can be obtained from the McCabe & Associates' web site, www.mccabe.com.

Above all, McCabe & Associates bring 23 years of experience and research in the area of source code metrics and baseline code analysis. Through documentation and training, McCabe IQ pinpoints the relevance and applicability of available metrics to the challenges of management decision-making.

McCabe SOFTWARE

Appendix A.
McCabe Metrics by Category

The following table contains a list of the primary predefined and derived metrics returned by McCabe IQ. The category (or categories) of software characteristics to which each metric belongs is indicated.

Metrics	Description	Size	Quality - Comprehensibility	Quality - Maintainability	Quality - Reusability	Risk	Readiness
Actual Complexity	Count of paths tested						X
Branch	Count of decision branches	X					
Code	Count of source lines	X	X				
Comments	Count of comment lines		X	X			
Coupling between objects	Count of distinct non-inherited classes on which a class depends		X	X	X	X	
Cyclomatic Complexity	Count of logical paths	X	X				
Cyclomatic Density	Cyclomatic Complexity divided by lines of executable code	X	X				
Depth of Inheritance	Maximum number of steps from the class node to the root of the tree and is measured by the number of ancestor classes		X		X		
Essential Complexity	Measure of structured-ness of the code		X	X			
Essential Density	Essential Complexity normalized by Cyclomatic Complexity			X			
Global Data Complexity	Quantifies the complexity of a module's structure as it relates to global and parameter data				X	X	
Global Data Density	Global Data Complexity normalized by Cyclomatic Complexity				X	X	
Lack of Cohesion	Measures of the dissimilarity of methods in a class by instance variable or attributes		X	X	X	X	
Number of Children	Count of immediate subclasses subordinate to a class in the hierarchy			X	X	X	
Number of Parents	Count of classes from which a class is derived			X	X	X	

Operands	Count of the operands in the code	X					
Operators	Count of the operators in the code	X					
Pathological Complexity	Measure of extremely unstructured coding behavior		X	X		X	
Percent Comments	Comments divided by the source number of lines		X	X			
Public Access	Count of access for class's public and protected data			X		X	
Public Data	Percentage of public and protected data in a class			X		X	
Response for Class	Count of the set of all methods that can be invoked in response to a message to an object of the class or by some method in the class		X	X			
Tested Branches	Count of branches tested						X
Tested Lines	Count of source lines of code tested						X
Weighted Methods per Class	Count of the methods implemented within a class			X	X		