Databorough

Complexity Metrics and Difference Analysis for better Application Management

Steve Kilner

WHITE PAPER

Table of Contents

Executive Summary.....1
Concepts.....2
  The Science Behind Software Maintenance.....2
  Why Audit and Metric Capabilities are Critical for Managing Legacy Applications.....5
Overview.....8
  Aspects of Quality in Software Maintenance.....8
  Translating Quality Into Measurable Items.....10
  How Measurable Items Become Actionable.....13
  Uses of Historical and Time Series Information.....14
  Version Comparison.....15
  Testability.....18
Use Cases for Metrics Reporting and Difference Analysis.....21
  Find the Most Complex Code in My System.....21
  Reducing Size of Application and Maintenance Workload by Removing unnecessary Code.....23
  Improving Project Management through Better Information.....24
  Cleaning Up your System to Recompile in its Entirety.....26
  Targeting Top 1% of Code that makes your JOB Difficult.....27
  Finding Programs most likely to Produce Defects when Modified.....28
  Identifying Unseen Risk in your Application.....29
  Monitoring Changes in Program Complexity to preserve System Value & Extend its life.....30
  Analyzing Metrics Time Series Data for Changes in System Complexity.....31
  Analyzing Differences in Source Code and System Objects in different Versions.....33

Executive Summary

The most challenging task in IT programming is maintaining and enhancing existing applications. This in fact represents the majority of worldwide programming budgets.

Unlike new software development, maintenance work is significantly impacted by characteristics of the software being modified. Modifying existing code can be exceptionally difficult and prone to cost overruns, delays and defects.

This paper discusses how you can improve your maintenance results by gaining quantifiable, measurable insights into your existing application. You can get significant information for these kinds of questions:

- How difficult will it be to modify this program?
- This program is very complex to modify, should we look for an alternative design?
- How difficult will it be to test this program if we modify it?
- Where are there risks that my programmers are not seeing?
- Do my programmers' estimates line up with the complexity of the programs?
- Is this program too complex to give to a junior programmer?
- The system is becoming more and more complicated, what's the best approach to simplifying it? Where do we start?

Many System i applications exceed a million lines of code. Over the span of their lifetime the systems become more and more complex, seriously, and adversely, impacting IT software projects and business objectives.

This paper discusses how to measure that complexity so you can act on it to lower your costs, increase your throughput and improve your quality.

"***You cannot manage what you do not measure.***"

\- Bill Hewlett, Hewlett-Packard

Concepts

The Science Behind Software Maintenance

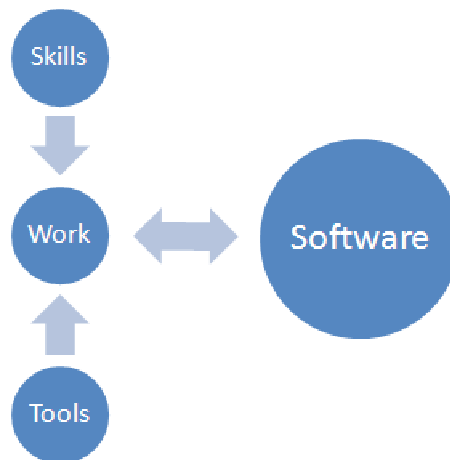
The ISO Software Quality Model defined in 1996 under 9126 and updated in 2005 under 2500n defines the means to measure the quality of a software application with six main quality characteristics:

- Functionality
- Reliability
- Usability
- Efficiency
- Maintainability
- Portability

Of particular importance to managers of legacy applications is that section called “Maintainability” which can be broadly defined as the ability to make changes for improving functionality, improving performance, meeting compliance requirements or fixing defects. The Model defines four characteristics that describe in more detail how maintainable a software system is:

- Maintainability
    - Analyzability – the ability to locate and scope features or faults within the code
    - Changeability – the effort required to make changes to the software
    - Stability – the likelihood that changes to the software will result in defects
    - Testability – the effort required to test changes to the software

Independently of project specifics, these characteristics of the software work in concert with programmers’ skills and their tools to determine how well the IT organization performs its role of supporting and enhancing applications.



The primary factors in the success or failure of software maintenance tasks are the programmers' skills, tools and the traits of the software being maintained.

The Human Factor - "It is harder to read a program than write it."

This familiar-sounding adage also sounds suspiciously like folk wisdom, but in fact there is serious science behind it. For nearly 20 years the *IEEE International Conference on Program Comprehension* has been meeting to research and discuss the challenges of maintaining software applications.

Two of the key topics in this subject area are:

- The mental processes people use to understand software
- The characteristics of software that make it easy or difficult to understand

The ISO Software Quality Model described above addresses the second of those points by stating that critical aspects of software quality are its analyzability, changeability, stability and testability. While all of these characteristics ultimately involve mental processes of people, they also lead to the hope that they that can be measured in themselves and thus, fit into a quality management program, which in turn should lead to increased productivity, programming throughput and higher quality.

How then, can one measure analyzability? There is no doubt that there are certain programs that, upon a little examination, lead one to quickly say, "This is very complicated. I do not want to maintain this program!"

An experienced programmer may look at a program and come to that conclusion in less than 60 seconds.

How does a programmer quickly assess the analyzability of a program?

That programmer is making a quick judgment on how much effort is required to build mental models of control flow and data flow sufficiently complete and

accurate to make software changes with an appropriate degree of confidence.

What did the programmer look at to make that judgment?

The Software Factor - Over the past four decades a number of formulas and models have been developed that attempt to measure the complexity of software by analyzing the source code. If these measurements are successful then it they will give us a good understanding of all those maintainability characteristics.

What do these complexity models measure?

Essentially they measure the things that are used in the mental processes and tasks of a programmer who is trying to understand a program:

- Build a mental model of the control flow of the program; i.e, the sequence of events and their conditioning.
- Build a mental model of the data flow of the program; i.e., what data goes in, how it's transformed, and what goes out.
- Map real world actions to actions observed in the code; e.g., "this is where we give a discount to frequent customers".
- Engage in "feature location", whereby the programmer is trying to find the code that implements features that are relevant to the modification task.
- Create and test out code modification hypotheses; i.e., "design" and "impact analysis".
- Utilize "beacons" to do all of the above; i.e., scan code and comments for keywords that signify relevance; e.g., a subroutine named WRITExxx probably outputs some data.
- Utilize "chunking" to gradually aggregate understanding of small pieces of code into large and larger pieces.

Some of these processes are more measurable than others:

Control Flow – the actual control flow of a program is determined by the control operations such as IF, DO, ELSE, etc., as well as the sequence of statements. If we can measure the number and complexity of control flow statements, plus the overall number of statements we can gain some insight into how challenging the task is to learn a given program for the purpose of modifying it.

Data Flow – the data flow of a program is determined by files that are input, fields that are transformed and files that are output. If we can measure the number and complexity of such fields we can gain some insight into how challenging the task is to learn a given program for the purpose of modifying it.

Map real world actions, feature location and beacons – you may wonder how an earth these things could be measured, but in fact there are some indicators we can use. Researchers have shown many times that well placed, well written comments and informatively named program tokens can greatly improve program comprehension.

Chunking – Code that is well organized and structured into loosely coupled, cohesive, visually distinct blocks is easier to mentally aggregate and comprehend than piles of spaghetti code.

Databorough's X-Audit tool provides metrics for many of these characteristics as this paper describes in detail.

Why Audit and Metric Capabilities are Critical for Managing Legacy Applications

Consider these two facts:

- 75% of worldwide IT programming budgets are dedicated to maintaining an enhancing existing software applications (Forrester Group)
- 40-60% of maintenance programmers' time is spent simply trying to understand the code they are working on (Software Engineering Book of Knowledge)

If you put those two facts together you come to the conclusion that the single most expensive task in all of IT programming is programmers trying to understand code.

What are the impacts on IT and businesses of this maintenance challenge?

Costs are high: it is more expensive to deliver a given amount of functionality to the business if it must be part of an existing application than if it is a new application

Expenses diverted to the old rather than the new: the bulk of IT programming budgets go to maintaining existing applications rather than developing new applications that could more quickly provide competitive advantages

Business opportunities missed: new business opportunities are missed or delayed because IT cannot respond quickly or cost-effectively enough to enhance existing systems to support new business opportunities

Operational and financial risks: changing highly complex, existing systems can introduce production defects that pose operational or financial risks

Threat of non-compliance: the business risks not meeting regulatory requirements in a timely manner if systems cannot be enhanced quickly enough

Why is it difficult to understand existing code?

At a very basic level there are two things involved, the programmer and the code. Programmers may be under-equipped, for whatever reason, to do the job, and that makes it difficult for them. Or, the code is in fact very complicated, and somewhat defiant of human comprehension.

What can be done to improve maintenance value delivery?

In his book examining over 12,000 software projects and their critical success and failure factors, Applied Software Measurement: Global Analysis of Productivity and Quality, long-time software metrics guru Capers Jones provides some insightful numbers from his analysis of maintenance productivity and quality.

The following table shows factors that positively impact maintenance productivity, and factors that negatively impact maintenance productivity.

Positive Factors	Impact%	Negative Factors	Impact%
Staff are maintenance specialists	+35	Error-prone code	-50
High staff application experience	+34	Embedded variables, data	-45
Table driven variables	+33	Low staff experience	-40
Low complexity code	+32	High complexity code	-30
Static analysis tools	+30	No static analysis tools	-28
Code Re-factoring tools	+29	Manual change control	-27
Complexity analysis tools	+20	No defect tracking tools	-22
Automated change control	+18	No quality measurements	-18
Quality measurements	+16	Management inexperience	-15
Formal code inspections	+15	No code inspections	-15
Regression test libraries	+15	No annual training	-10

Like many such analysis, some of the good and bad factors are just the flip side of each other, but here is what stands out and should be heeded by the thoughtful IT manager:

The dominant factors that affect maintenance productivity, costs and quality, both good and bad, are related to the complexity and quality of the code, and the tools available to deal with them.

Here is another view of that table highlighting the relevant factors, and the solutions that Databorough delivers to directly address those factors.

Positive Factors	Impact%	Negative Factors	Impact%
Maintenance specialists	+35	Error-prone code (X-Audit)	-50
High staff experience	+34	Embedded variables (X-Analysis)	-45
Table driven variables (X-Analysis)	+33	Low staff experience	-40
Low complexity code (X-Audit)	+32	High complexity code (X-Audit)	-30
Static analysis tools (X-Analysis)	+30	No static analysis tools (X-Analysis)	-28
Code Re-factoring tools (X-Redo)	+29	Manual change control	-27
Complexity analysis tools (X-Audit)	+20	No defect tracking tools	-22
Automated change control	+18	No quality measurements	-18
Quality measurements	+16	Management inexperience	-15
Formal code inspections	+15	No code inspections	-15
Regression test libraries	+15	No annual training	-10

How can you start achieving these kinds of gains in productivity and quality?

Very simply, you need better information for management and better information for programming.

Databorough supplies two essential tools to improve productivity and quality for maintenance operations that directly address the above statistics as found in over 12,000 software projects:

X-Analysis – An application cross reference and static analysis tool that enables managers, systems analysts and programmers to rapidly and thoroughly research existing applications in support of application enhancement, debugging and documentation tasks.

X-Audit – *The focus of this paper* - is a source code and object analysis system that provides metrics, alerts and time series comparisons of the state of your application to enable you to focus attention on the areas of your system most in need of correction, improvement or attention.

With this information available you can begin to answer some truly important questions:

- How can I find the most complex code in my applications?
- Can I reduce the size of my applications, and thereby the maintenance workload, by removing unnecessary code?
- How can I improving my project management, estimating, scheduling, budgeting, testing, etc., through the use of this information?
- How can I clean up my applications so they will recompile in their entirety?
- Is there a way to target the top 1% of my code that makes our job the most difficult?

See the sections on *Popular Use Cases* for more examples and detailed information.

Overview

Aspects of Quality in Software Maintenance

As the earlier section, *The Science Behind software Maintenance*, describes, the ISO Software Quality Model breaks down software quality into six characteristics, one of which we are most concerned with as managers of legacy systems (shown here broken down further):

- Functionality
- Reliability
- Usability
- Efficiency
- Maintainability
    - Analyzability – the ability to locate and scope features or faults within the code
    - Changeability – the effort required to make changes to the software
    - Stability – the likelihood that changes to the software will results in defects
    - Testability – the effort required to test changes to the software
- Portability

In this paper we are specifically concerned with software maintenance and how we can obtain useful quality information by analyzing source code and other system information. And even more specifically, we are concerned with how we can quantify that information by casting it into *a framework of metrics*.

But let's first look in another direction and think about another set of ISO standards, those that pertain to Software Maintenance. *ISO 14764, Software Life Cycle Processes for Maintenance* describes four categories of maintenance activities:

- Corrective – fix defects
- Adaptive – modify the software to keep it useful i.e. enhancements
- Perfective – improve either the performance or maintainability of the software
- Preventive –preemptively detect or correct latent defects in the software

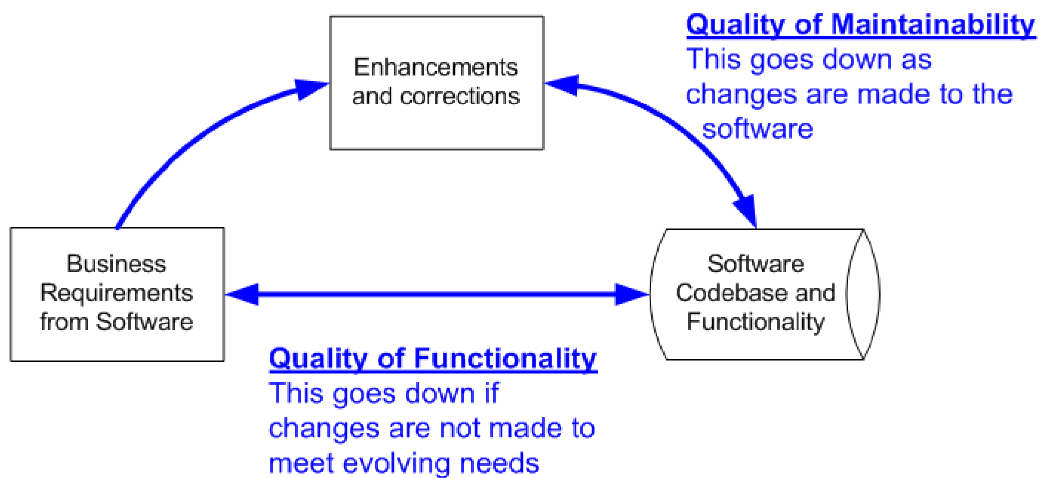
Various studies have shown that upwards of 80% of total activity is adaptive, in other words, enhancements to the system. There is sometimes a view that most of the work is corrective, but it has also been shown that many tasks presented by users as bug fixes are in fact requests for changes in functionality. Many maintenance organizations do not fully distinguish between corrective and adaptive activities and often switch staff freely between these types of tasks.

Key Principal: All Software Quality Declines Over Time

However the work is categorized and managed, over time, the quality of the software goes down. In fact, unless actions are taken to correct it, it is *completely unavoidable* that the quality of the software goes down over time:

- If the software is maintained without full regard to maintainability it will necessarily become more complex, and thus its maintainability quality will decline, or

- If the software is not maintained it will necessarily become less useful to the evolving user organization, and thus its functionality quality will diminish



The evolution of software systems over time has been studied by a number of researchers and academics. Professor Meir Lehman of Imperial College London identified a number of observations of how software evolves over time in what is often called The Eight Laws Of Software Evolution. For the IT manager with a big picture of the forces at work in software maintenance it is worth having some awareness of these forces:

1. Continuing change – software must be continually adapted or it will become less and less satisfactory
2. Increasing complexity – as software is changed it becomes increasingly complex unless work is done to mitigate the complexity
3. Relationship to organization – the software exists within a framework of people, management, rules and goals which create a system of checks and balances which shape software evolution
4. Invariant work rate – over the lifetime of a system the amount of work performed on it is essentially the same as external factors beyond anyone's control drive the evolution

5. Conservation of familiarity – developers and users of the software must maintain mastery of its content in order to use and evolve it; excessive growth reduces mastery and acts as a brake
6. Continuing growth – seemingly similar to the first law, this observation states that additional growth is also driven by the resource constraints that restricted the original scope of the system
7. Declining quality – the quality of the software will decline unless steps are taken to keep it in accord with operational changes
8. Feedback system – the evolution in functionality and complexity of software is governed by a multi-loop, multilevel, multiparty feedback system

Why is this important, or how is it useful?

The job of most IT managers is typically to *get it done faster, better, cheaper.* ("pick two," as the saying goes) Often unstated is the further directive to continually improve in those measurements. Not just today, but next year, and the year after.

But implicit in all of the above is that *much of what you do today will slow you down tomorrow*. Unless, that is, you take action on the implicit advice of the second law and do work to maintain your system's maintainability.

And indeed, many IT organizations with a long view of the life of their software and its responsiveness to business needs take proactive steps to

maintain maintainability
and
manage to maintainability

But how is that possible? How do you undertake a program of maintaining maintainability and managing to maintainability?

For that, we return to the wisdom of Bill Hewlett:
"You cannot manage what you do not measure."

Translating Quality Into Measurable Items

Again, this paper concerns itself with what aspects of quality that can be measured by analyzing source code and other system information. What aspects of quality cannot be measured this way? We cannot, for example, measure how well the system functionality meets business needs, since we have no way in the system to measure business needs. We can also do very little to measure system reliability – though we could perhaps measure system availability, measuring defects calls for a tool designed for that purpose.

What can we measure by looking at the source code and system objects?

As mentioned earlier, there are some key mental processes that programmers engage in when performing maintenance. If we can measure things that relate to these processes we will get some understanding of the level of maintainability quality:

> **Control Flow** – what conditions control the program's operations and what is their sequence?
>
> **Data Flow** – what are the files and fields that are input, how are they transformed, how are they output?
>
> **Map real world actions, feature location and beacons** – what is the quality of names assigned to program tokens and the level of commenting?
>
> **Chunking** – to what degree is the code loosely coupled and cohesive and readable?

If these are the mental processes that impact maintainability, what be measured for them?

Looking at this in strictly RPG terms we can define a number of aspects of the source code that can help us measure these characteristics:

RPG Metrics that indicate comprehensibility of *Control Flow*

- Cyclomatic complexity – basically a count of Ifs, Dos, FORs, WHENs, etc.
- Greatest depth of nested ELSEs.
- Number of GOTOs or CABxxs.
- Greatest depth of nested IF/ Dos.
- Greatest number of statements in an IF/DO block.
- Greatest depth of loops within loops.
- Greatest number of statements in a subroutine.
- Depth of subroutine calls.
- Uses RPG Cycle for processing.
- Number of statements with conditioning indicators.
- Decision density.
- Number of delocalizing statements.

RPG Metrics that indicate comprehensibility of *Data Flow*

- Halstead volume – basically a measure of the number of distinct fields and their uses
- Number of database files
- Number of device files
- Number of EXFMTs/ READs to display files
- Number of display file formats with fields that output to a database file
- Number of sub-files in program
- Number of called programs

- Number of calling programs
- Number of fields whose value was set
- Number of fields whose value was used
- Number of global fields whose value was set
- Number of global fields whose value was used
- Number of files updated
- Number of program-described input files
- Number of program-described output files
- Number of applicable OVRDBFs
- Number of applicable OPNQRYF statements
- Average variable span by line numbers
- Total variable span by line numbers
- Average variable span by subroutine count
- Total variable span by subroutine count
- Number of delocalizing statements

RPG Metrics that indicate comprehensibility through *Knowledge Mappability*

- Number of non-hyper-local field names of less than x characters
- Number of lines of comments

RPG Metrics that indicate comprehensibility through Chunkability

- Number of actual lines of code
- Number of actual lines of comments
- Greatest number of statements in a subroutine
- Greatest number of statements in an IF/DO block
- Number of implicit global parameters in a procedure
- Number of delocalizing statements
- Maintainability index – a formula developed by HP through experience
- Number of /COPY members
- Number of statements changed/added in the last 30-60-90-180-360 days
- Number of months in the last 12 months that had one or more statements added/changed

Some of these metrics are useful in more than one category and some do not fit neatly into these categories or are not perfect indicators, but nevertheless, it should be clear that there are in fact a number of useful metrics for understanding maintainability and overall program complexity.

It should also be clear that these metrics can in fact be computed from typical source code, and in fact, that is precisely what Databorough's X-Audit tool delivers.

If you are an experienced programmer who is managing a large application, you may look at this list and nod your head in recognition that many of these things would be interesting to have in a sortable list.

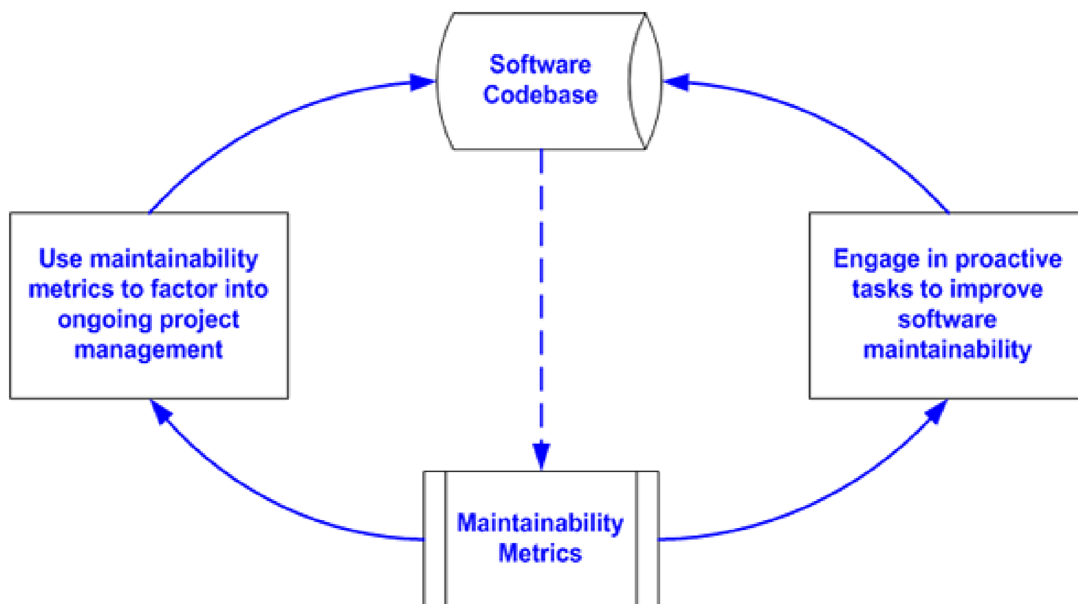
But the real question is, ***how can these metrics make a meaningful difference?***

How Measurable Items Become Actionable

"What gets measured is what gets done."

- Tom Peters

The following diagram shows the two primary ways in which software metrics can help manage a software maintenance operation.



The left box is meant to show that metrics information can be used to bring better management and planning to your software projects. Some of the ways this information can be used are:

- Adjust programming estimates, and therefore schedules and costs
- Decide where more thorough analysis is necessary
- Decide which resources are most appropriate for a task
- Develop more appropriate and detailed testing plans.
- Advise the business of additional project risks
- Decide on alternative design plans to minimize changes to highly complex code

For more information on how to use metrics for these purposes see the use case *Improving Project Management Through Better Information.*

The right box is meant to show that metrics information can be used help you keep your software in a more maintainable state and thus preserve its long term value and ability to respond to business needs quickly and cost-effectively.

This type of work can be analyzed in a couple ways, leading to tasks that:

- Refactor programs that cross a certain threshold of complexity, or,
- Refactor programs that have shown a large increase in complexity and are expected to continue to do so

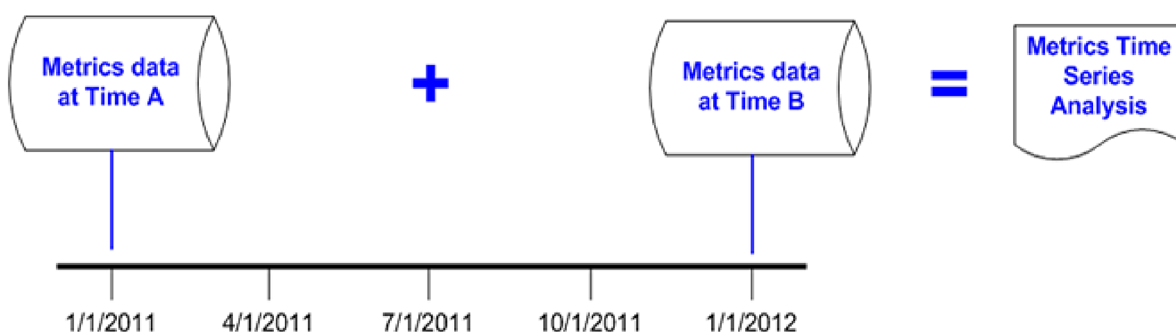
For more information on maintaining maintainability see the following use cases:

- Monitoring changes in program complexity to preserve system value and extend its useful life
- Targeting the top 1% of code that makes your job difficult
- Finding programs most likely to produce defects when modified
- Identifying unseen risks in your application
- Cleaning up your system so it will recompile in its entirety

Uses of Historical and Time Series Information

The metrics discussed so far have been point in time metrics, in that they analyze source code and system objects at the time the metrics data is generated. For overall system management there are other useful perspectives that involve the dimension of time and change.

One important perspective comes from understanding the change in the complexity and maintainability of your system over time:



In this case metrics data collected at two or more different points in time are compared and the differences are shown.

Some of the purposes of this sort of analysis are:

- Determine the overall success of maintaining maintainability

- Identify programs that cross a defined threshold of maintainability into unmaintainability and are thus candidates for Refactoring
- Identify programs with sudden changes in complexity and that are forecast to continue with that trend, and are thus candidates for Refactoring or other attempts to keep maintainable
- Identify increases in complexity where they were not expected, as a possible indication of poor programming or design

See the use case *Analyzing Metrics Time Series Data for Changes in System Complexity* for more information.

Version Comparison

Version comparison is a facility that enables you to compare two different versions of your application at both the source code and object levels. Here are a few common scenarios where this is useful:

- Compare a version of the application in use in one location to the version in use at another location
- Compare a new version of a packaged product release to the version currently installed in order to understand the differences
- Compare the current state of the application to the state it was in at a point in time in the past

Difference Analysis

A product such as Databorough's X-Audit can do these comparisons and give detailed reports on both source and object differences between the versions.

This information can point to changes that have to be made to bring two versions into harmony, or to integrate a new version of the source. By comparing versions from different points in time the analysis can reveal unexpected changes in the system in the interim.

Information contained in such an analysis includes:

- Files and programs that have been added, changed or deleted
- Fields whose attributes have changed
- Changes in database relationships and dependencies
- Business rules that have been changed, added or deleted
- Source statements that have been changed, added or deleted

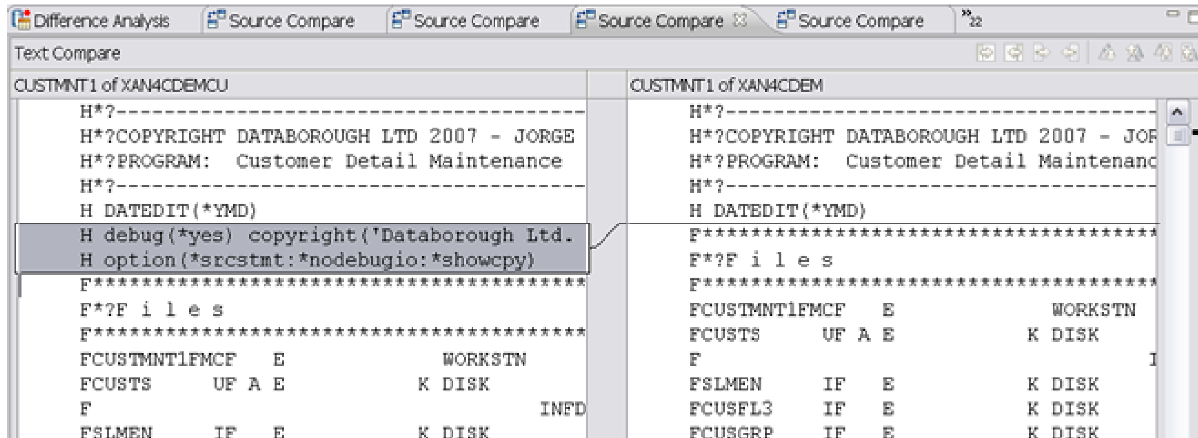
Source Comparison

The last type of analysis in the above list can become very involved as potentially many source members may have been changed. It is important that a facility be available to quickly drill down from a changed source member to the specific lines of code that have been changed, added or deleted.

A source comparison tool is essential for analyzing the differences in source code between the versions being compared. A good tool should show you:

- Which source members have been changed and allow you to drill down into:
- Which source statements have been changed, added or deleted

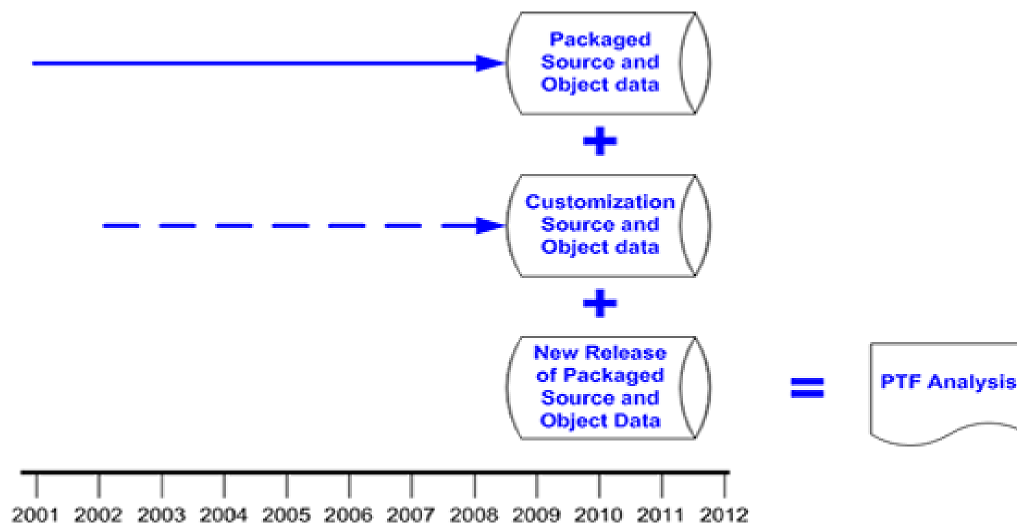
Here is an example of a source comparison; in this case two H specification statements exist in the left hand version which do not exist in the right hand version:



PTF Analysis – A Special Case of Version Comparison

If you are using a packaged software application that you have customized to meet your needs then you will probably have encountered the challenges that come when the vendor provides a new release of the product. How do you integrate your past changes with the new version of the software? What have you changed? What have they changed?

This is in fact a serious challenge and potentially a great deal of analysis work. The following diagram depicts this situation.



In this case an analysis of the source and objects in the new release of a packaged software product (bottom) are compared against the source and objects that have been customized in the past (middle) and the current base installation of the package (top).

This sort of analysis can be quite labor intensive but the use of a tool like Databorough's X-Audit PTF Analysis can save a great deal of time and prevent the risk of mistakes.

The following types of conditions are analyzed and reported on. In these examples "PTF library" refers to the new release of package changes and "customized" library refers to the customizations that have been made over time to the base package.

Modified - The object from the PTF library was found in one of the customized libraries. The PTF object will have to be reviewed and changes applied in the customized library must be manually applied to the object in the PTF library.

New - The object from the PTF library was not found in the base repository. The PTF object can be placed in the base library.

Apply - The object from the PTF library was found in one of the base libraries but not in any of the customized libraries. Therefore the PTF object can overlay the object in the base library.

Refers - The object from the PTF library refers to one or more objects in one of the customized libraries. The PTF object will have to be analyzed to make sure all customized objects referred to still meet the requirements of this object.

Referenced - The object from the PTF library is referenced by an object in one of the customized libraries. The customized objects will have to be reviewed to make sure the PTF object will still interface properly to the customized objects.

Testability

Testability is one of the characteristics of Maintainability, which, again, is one of the ISO characteristics of software quality.

Testability and Metrics

Most metrics that pertain to complexity and maintainability, also pertain to testability. If a program is more complex, and more difficult to maintain, it tends to be more difficult to test. With perhaps a few exceptions, pretty much all of the metrics in the section Overview: Translating Quality into Measurable Items impact a program's testability.

Improving Testability With Tools

Reducing code complexity can bring some relief in terms in testability, but more likely to make a more dramatic and immediate impact on testability is the use of tools.

Managing Code Complexity for Testability – Control Flow

The completeness of test plans is often measured in terms of coverage. There are several levels or dimensions of coverage to consider:

Function, or subroutine coverage – measures whether every function or subroutine has been tested

Code, or statement coverage – measures whether every line of code has been tested

Branch coverage – measures whether every case for a condition has been tested, i.e., tested for both true and false

Loop coverage – measures whether every case of loop processing has been tested, i.e. zero iterations, one iteration, many iterations

Path coverage – measures whether every possible combination of branch coverage has been tested. Large programs can have huge numbers of paths through them. A program with a mere 20 IF, DO or WHEN statements can have over one million different paths through it (paths = 2^n).

Removing redundant conditions, and organizing necessary conditions in the simplest possible way help to minimize control flow complexity and thus minimize both the probability of defects and the required testing effort.

Managing Code Complexity for Testability – Data Flow

Also of concern for managing testability is the impact of code implementation on the complexity of data flow. This type of complexity can be measured in a few

different ways:

Depth of transformation – A variable that is moved from an input file directly to an output file is said to a transformation degree of 1. If it is first multiplied by 10, for example, the degree is then 2. The more that data is transformed the more complex the test plans must be.

Dispersion, or span of modification – If the statements that modify a given variable are scattered around a program it will both be more likely to have defects and more likely to require more testing. If a given variable is set three times in the span of ten consecutive statements that is much less likely to produce defects or testing challenges than if the variable is modified three times each in different subroutines separated by 1,000 lines of code.

By considering these data flow complexity factors when designing the program code the ultimate testability and quality of the program can be increased.

Using Tools To Improve Testability

Tools can be of great assistance in the testing effort, bringing gains in both productivity and quality. Examples of tools are:

Complexity metrics – as this paper discusses, understanding the complexity metrics of a program to be tested helps in preparing both project plans and testing plans. See the use cases *Improving Project Management Through Better Information and Finding Programs Most Likely To Produce Defects When Modified for more information*.

Generation and Validation of Test Plans – see the section immediately below for more information on this.

Tracking code and branch coverage – tools can be of great assistance in tracking whether all statements and conditions in a program have been tested.

Generation and Validation of Test Plans

A common method of developing a test plan is to follow a hierarchy as follows:

- Business Processes
  - Test Cases
    - Test Scenarios

In System I applications a given interactive program might typically be thought of at the test case level and have any number of individual test scenarios.

A very useful approach to developing the test case and test scenarios is to translate the program into a UML Activity Diagram. This kind of diagram shows all the different use paths a user can follow in executing the program and provides an excellent foundation for the test scenarios. (note that these paths are not exactly the same thing as the code paths described above, though they are obviously related).

Shown below is an example of a portion of an activity diagram as produced by X-Analysis which can be used to improve testing productivity and quality.

In the above diagram from Databorough's X-Analysis UML feature, each connector would typically be designated as a test scenario, with conditions, data, actions and results defined for that function.

Use Cases for Metrics Reporting and Difference Analysis

Find the Most Complex Code in My System

Why it's important and valuable

There are three categories of reasons for why this is valuable information:

1. Project planning – With complexity metrics you can make more fine-grained judgments about the strategy and planning of your projects. See the use case about improving project management for detailed information.
2. Proactive complexity mitigation – IT managers with a long term view of their system's health take proactive measures to prevent their code from becoming excessively complex. See the use case about extending the life and value of your system for more information on this perspective.
3. Design recovery and migration – If you are extracting business rules or migrating your code to another language you may want to plan on manual, corrective activity to deal with overly complex code.

What information is needed and why

In this use case example we will use either or both of the basic complexity reports that are pre-configured in X-Audit:

1. COMPLEXP – metrics by program, or
2. COMPLEXS – metrics by subroutine

Both of these reports have the same data except that the latter also has subroutine names, giving more detailed results. Otherwise, both of these contain the same metrics:

- Number of actual lines of code
- Greatest number of source records in a subroutine
- Greatest number of statements in an IF/DO block
- Cyclomatic complexity
- Halstead volume
- Maintainability index
- Number of virtually global variables
- Total or average variable span by line number or subroutine
- Decision density
- Number of database files

- Number of called programs
- Greatest depth of nested IF/ DOs
- Greatest depth of nested ELSEs
- Number of GOTOs or CABxxs
- Greatest depth of loops within loops
- Decision density

These reports both select all objects with an attribute of RPG or RPGLE.

A little bigger picture: X-Audit provides a number of metrics for evaluating complexity. There are three ways to think about measuring the complexity of your code:

1. Using traditional, cross-language metrics, such as Cyclomatic Complexity, Halstead Volume and Maintainability Index.
2. Using additional metrics provided by X-Audit that are more language and System i specific.
3. Using your own custom metrics:
    A) Computed by you using the provided X-Audit formula, which enables you to combine either of the above metrics.
    B) Writing your own code analysis programs and creating your own application-specific metrics using the X-Audit user exit program facility.

Evolving the Most Representative Metrics

Eventually you will want to decide on which metrics best represent complexity in your application. These might be one or more of the pre-packaged metrics or some combination of them that you perform your own customized computations on.

How to generate the report

Select either of the above reports and click on Run Metrics Report on the main X-Audit screen.

If you want to modify either of these reports you can make a copy of it and change any of the parameters.

Analyzing the Results

The first time you see the results of this report you will realize how much measurable information you've been missing. You will want to play with the data in a number of ways to develop a model of which metrics give you the best indication of your own system's complexity. Here is an example of a screen-shot for a shorter version of the above reports:

Obj	BaseComplex	MaintIdx	DecDensity	VarSpan	LOC
AT201R2	94	94	100	109	3,384
AT144R	88	83	77	86	3,784
AT200R	88	85	92	96	5,984
AT198R	84	85	79	78	6,720
AT192R	83	93	91	82	6,142
AT156R	71	81	80	72	4,899
AT201R	56	46	52	56	3,024
AT110R	24	17	17	21	664
AT178R	23	19	12	4	255
AT112R	15	5	13	21	340

In this example all the metrics except Lines Of Code have been normalized to a scale of 1-100. Doing this helps read the results and also facilitates combining individual metrics into combined, weighted scores.

Also in this example note that the first metric, “Base-Complex”, is a custom, user-defined metric that is comprised of several other metrics that the user has decided most accurately convey complexity in this particular application.

Note that this is sorted by Base-Complex, and note how Lines Of Code does not correlate well to complexity. This has been shown by many studies over the years – Lines of Code is a poor indicator of complexity.

Reducing Size of Application and Maintenance Workload by Removing unnecessary Code

Why it’s important and valuable

Many systems accumulate dead objects or dead code with little apparent harm. The key word there is “apparent”. Many IT organizations waste an unknown number of hours maintaining, recompiling and testing objects that are no longer actually in use. Over a period of years the number of these objects tends to pile up, as does the amount of wasted effort.

What information is needed and why

In this use case example we will use either or both of the unused object reports that are pre-configured in X-Audit:

1. UNUSEDOBJ – unused objects, based on object description last used date
2. UNUSEDCOD – unused sections of code e.g., subroutines or procedures not called

How to generate the report

Select either of the above reports and click on Run Metrics Report on the main X-Audit screen.

If you want to modify either of these reports you can make a copy of it and change any of the parameters. (See section How the Product Works for more information on the screen options available to you)

Analyzing the Results

The two reports work very differently and lead to different tasks you will want to undertake to reduce your maintenance workload.

UNUSEDOBJ – this report looks at the last used date from System i object description data. Be sure you understand exactly how the last used date is set and reset on the System i for objects before archiving them.

UNUSEDCOD – this report lists subroutines and procedures that have zero calls to them within their program. These represent sections of code that can be deleted. Be sure you follow good source management procedures, including archiving, before deleting code.

Improving Project Management through Better Information

Why it's important and valuable

With complexity metrics you can make more fine-grained judgments about the strategy and planning of your projects. Complexity information can help you:

a) Adjust programming estimates, and therefore schedules and costs
b) Decide where more thorough analysis is necessary
c) Decide which resources are most appropriate for a task
d) Develop more appropriate and detailed testing plans
e) Advise the business of additional project risks
f) Decide on alternative design plans to minimize changes to highly complex code

What information is needed and why

This use case utilizes your evolved complexity metrics – the ones that best represent complexity in your system.

How to Apply Metrics to Project Management

<u>Improving Estimates</u>

Research studies have shown that presenting information to programmers about the program they will be working on materially affects their estimates of the work to be done. By supplying some facts to supplement their intuition and experienced based judgment, you can obtain more realistic estimates of the amount of effort involved. Examples of what can improve the quality estimates include:

- Number of calls to a subroutine to be changed
- Number of calls made to a subroutine
- Number of uses in a program of a variable to be changed
- Number of uses in a program of a file to be changed
- Cyclomatic complexity or other IF/DO metrics of code to be changed
- Number of files, input formats and/or subfiles in a program
- Number of statements in relevant programs, subroutines, or large IF/DO blocks to be changed

Decide Where More Thorough Analysis is Necessary

By understanding the complexity structure of a given program requiring modification, a manager can be sure a programmer has delivered a quality estimate by understanding what subroutines require changes and then comparing the programming estimates against the complexity metrics for those subroutines.

If the values of certain variables in the program will be affected then the manager can also examine how many uses of the variables exist in the program, thus understanding the potential impact of the changes, and the amount of impact analysis required to do a quality job.

For example, there is a large difference in the amount of analysis work required between adding a few lines of code to a simple section of the mainline that does not affect variables, and adding a few lines of code located in the middle of deeply nested IF/DO/ELSE blocks in a subroutine called from many places in the program, where those changes affect variables widely used throughout the program.

Without doing the code research itself, a managers have had few options for evaluating the estimates provided by programmers. By simply asking programmers which subroutines they will be modifying the manager can now evaluate the complexity metrics of those code sections and make a more informed judgment of whether the programmer’s estimate is sufficiently considered.

Decide Which Resources Are Most Suitable For a Task

For a given project, once the list of programs to be modified has been compiled, the IT manager can look at the metrics for the programs and decide which ones require the use of resources with either special program knowledge or the ability to handle highly complex programs. While most IT managers know this to some degree from experience, the availability of metrics presents the basis for a more quantifiable and consistent decision process.

Develop More Detailed and Appropriate Testing Plans

By understanding which subroutines are to be modified, and what their complexity metrics are, the project plan can be adjusted to account for additional testing for more complex sections of code being changed. Useful metrics include all of the Cyclomatic and IF/DO complexity metrics, as well metrics relating to numbers of files and fields involved.

Advise the business of additional project risks

Complexity metrics represent tangible information that IT can present to the business when explaining the challenges of particular projects. The metrics can be used to explain why some tasks require more time than others, and why some tasks are more likely to result in production defects.

By making complexity metrics a regular part of project plans presented to business stakeholders, IT can shape the overall process to be based more on facts rather than intuition and persuasion.

Decide on alternative design plans to minimize changes to highly complex code

For a given project, once the list of programs to be modified has been compiled, the IT manager can look at the metrics for the programs being changed and decide to investigate alternative design plans that might circumvent the most complex sections of code being changed. Obviously, all projects have more chance of success if they deal with the simplest possible code.

Cleaning Up your System to Recompile in its Entirety

Why it's important and valuable

Why this is important almost goes without saying, but it often becomes one of those things that is important but not urgent. What can make it more urgent is if you plan on doing something like any of the following:

- Install a new release of packaged software
- Re-engineer or migrate your system
- Execute a large application enhancement project

What information is needed and why

There are a number of "alert" type metrics provided with X-Audit that are useful for this purpose. Some of them directly indicate it is impossible to compile your system accurately, others indicate generally undesirable conditions that are worth investigating.

- No source for existing object
- Source was changed after object was created
- No object for existing source
- Logical file is duplicate of another
- Logical file is not used in any programs
- File has no members
- File is internally described
- File format level used in program does not match database file
- Program has hard coded libraries

How to generate the report

This is a pre-configured report provided with X-Audit – see the category, *Source/Object Reports*.

Targeting Top 1% of Code that makes your JOB Difficult

Why it's important and valuable

Numerous software studies have shown that the majority of defects come from a small percentage of programs, the majority of complexity in a system is contained in a small percentage of programs, maintenance tasks tend to revolve around a small percentage of programs, and so on. The Pareto Principle, aka, the 80-20 rule, doesn't apply, it's more like the 90-10 rule, or the 95-5 rule, or, like the title suggests, even the 99-1 rule.

Here's a formula worth considering:

(most complex code) U (most frequently changed code) -> (most troublesome, costly code)

In other words, the intersection of your most complex code and your most volatile code deserves some serious attention!

What is it that makes code both complex and volatile?

- Defect repair leads to changes
- Hard coding leads to changes
- Inadequate design vs. business or technical needs leads to changes
- Changes lead to ever increasing complexity
- Complexity leads to defects

And so on. Yes, there can be a vicious cycle at work.

If you can identify your most complex, volatile code what can you do about it?

- Remove hard coding.
- Revisit other design aspects and see if it needs to be upgraded.
- Have managed code walk through to inspect it for defects – various studies have put the cost of user-discovered defects at 10-100 times higher than developer discovered defects
- Refactor the section of code to simplify it; possibly break it into smaller, more manageable and more testable pieces.

What information is needed and why

The first use case, How Can I Find the Most Complex Code In My System showed you how to do just that. What is needed for this use case is to combine that information with information about what source code is changed and how frequently.

How to generate the report

Depending on what you have done regarding defining which metrics you want to use for complexity analysis, you may be able to simply add the X-Audit metric SRCCHG360 to your metrics report. Alternatively you can run the report Source Change Volatility under the category Source/Object Reports and export all results to spreadsheets where you merge and analyze the complexity and volatility metrics results.

X-Audit source volatility analysis works by analyzing source change dates in source files. This provides limited information. X-Audit also provides an interface for more detailed source change data that can be fed from your change management system.

Finding Programs most likely to Produce Defects when Modified

Why it's important and valuable

Knowing which programs are the most likely to produce defects when modified can help you:

- Seek alternative design solutions that avoid those programs
- Adjust your programmer resource plan to place your most reliable programmers on those challenging programs
- Allow for additional time and resources in project plans for more extensive testing
- Alert business users to increased project risks

- Decide to proactively refactor/redesign your programs

What information is needed and why

Most of the complexity metrics have some bearing on how likely it is that modifications will lead to defects for a given program, but certain metrics are generally more useful than others, in particular, those that relate to the difficulty of impact analysis, or indicate program volatility:

- Number of virtually global variables
- Total or average variable span by line number or subroutine
- Decision density
- Greatest depth of IF/DO/ELSE blocks, or GOTO count
- Depth of subroutine calls
- Number of called programs or external procedures
- Number of statements changed in the last year

How to generate the report

This is a pre-configured report provided with X-Audit. Under the category, RPG Complexity Reports, select and run the report, Defect-prone programs.

Analyzing the Results

By developing a practice of tracking defects and measuring them against these defect analysis metrics, or others that you develop over time, you can refine your ability to predict defect levels and plan accordingly.

Identifying Unseen Risk in your Application

Why it’s important and valuable

This topic divides into two categories of risks:

- Object level risks, related to system object management
- Code level risks, related to the code of programs

The reason why identifying risks is important is self-evident. Quantifying the potential costs of the risks is also important, more so for weighing the cost of the repair effort than for doing the analysis, which is as simple as running the report mentioned below.

What information is needed and why

There are many potential risk factors in a system, here are a few to consider:

- Programs have non-approved hard coded libraries
- No source code exists for an object
- The source code has been changed since the object was created
- The same field name is found in multiple files in a program
- RPG UPDATE operations are done without listing fields
- RPG WRITE operations exist for input/update files and no CLEAR operation is found

How to generate the report

This is a pre-configured report provided with X-Audit. Under the category, Source/Object Reports, select and run the report, Unseen Risks.

Monitoring Changes in Program Complexity to preserve System Value & Extend its life

Why it's important and valuable

The second law of software evolution states, "as a system evolves, its complexity increases unless steps are taken to reduce it." Or, as someone else said, "the act of maintaining software necessarily degrades it."

Your applications are an asset of your business. As you maintain them over time you cause the value to depreciate by making them more complex and less maintainable. Arguably you are also increasing their value by adding functionality, but there is no doubt that applications become more time-consuming and costly to maintain as they age.

Some IT organizations address this growing complexity by proactively maintaining maintainability. After establishing a set of metrics that best represents complexity for their applications, they periodically measure the complexity of the entire system. Programs that either cross a threshold of complexity or show large increases in complexity are candidates for refactoring.

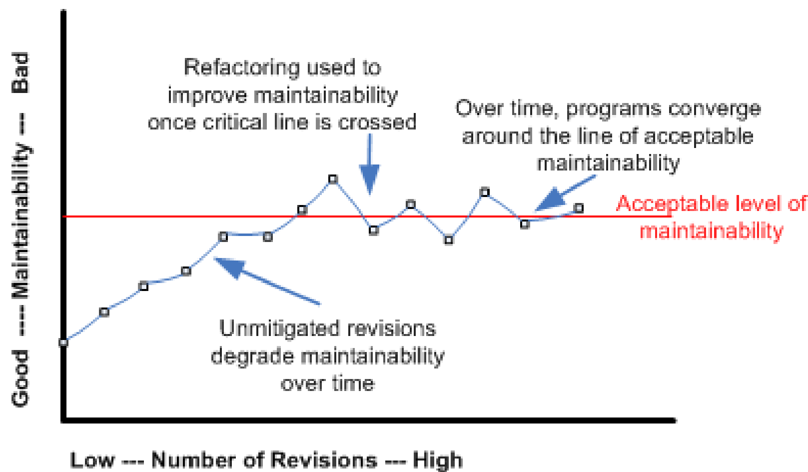
What information is needed and why

This process is based on the set of metrics established in the first use case, *How Can I Find the Most Complex Code In My System?* Armed with that information, there are two basic approaches:

- Refactor programs that cross a certain threshold of complexity
- Refactor programs that have shown a large increase in complexity and are

expected to continue to do so

The following diagram depicts the growth in complexity of a particular program and shows that when it crossed a defined threshold of complexity it was Re-factored to preserve its maintainability:



It is useful to store metrics in order to compare them to future values of the metrics. As the chart shows, analyzing the differences can reveal important patterns of complexity as it relates to overall analyzability, changeability, stability and testability.

This information should also be reviewed with information about past program volatility and known plans for future projects.

Analyzing Metrics Time Series Data for Changes in System Complexity

Why it's important and valuable

There are at least two good examples of the benefits available by examining changes in complexity metrics over a period of time:

- Patterns in complexity growth and system growth that are obscured in the hurry of day-to-day work can be seen and future development plans can be adjusted or created based on the new understandings
- Observed increases in complexity that do not match expectations can reveal poor design or programming practices, which in turn may lead to corrections, better training or adjustments in future resource assignments

What information is needed and why

Obtaining this time series information is simply a matter of storing metrics at

different points in time, calculating the differences and reporting on them. This is best done when an organization has identified the specific metrics that give the best indication of complexity and maintainability for the application.

Here is an example of a baseline metrics report at a given point in time. In this case a baseline complexity metric has been customized by the user and the report is sorted top to bottom in that sequence. Also, all metrics scores have been normalized to a scale of 1-100.

Obj	BaseComplex	MaintIdx	DecDensity	VarSpan	LOC
AT201R2	94	94	100	109	3,384
AT144R	88	83	77	86	3,784
AT200R	88	85	92	96	5,984
AT198R	84	85	79	78	6,720
AT192R	83	93	91	82	6,142
AT156R	71	81	80	72	4,899
AT201R	56	46	52	56	3,024
AT110R	24	17	17	21	664
AT178R	23	19	12	4	255
AT112R	15	5	13	21	340

At a later point in time we run the analysis again and get a similar set of results, but the metrics are now different.

The following report shows which metrics have changed and by how much. A positive number indicates the metric value has increased.

Time Series Report Showing Changes In Metrics

Obj	BaseComplex	MaintIdx	DecDensity	VarSpan	LOC
AT192R	10	-3	1	-1	403
AT156R	9	-1	9	0	364
AT201R	9	-4	3	3	241
AT201R2	7	-5	2	2	176
AT200R	7	-3	1	6	228
AT198R	5	-1	5	7	-273
AT110R	3	0	-1	3	-36
AT112R	3	3	8	0	26
AT178R	1	0	7	5	48
AT144R	-1	4	6	0	106

In this example the program with the largest increase in the base complexity metric is listed first, showing an increase of 10. Correspondingly its maintainability has dropped by 3 and the number of lines of code have increased by 403.

In this example what also stands out for the IT manager is that program AT201R

has had a substantial increased in base complexity of 9, yet the IT manager knows that he had only asked for a simple change to this program – that is worth investigating.

Analyzing Differences in Source Code and System Objects in different Versions

Why it's important and valuable

There are a number of circumstances where it is useful to compare different versions of an application:

- A software vendor delivers a new release of the application and you need to know what has changed so you can confirm your customizations or interfaces will work correctly
- You operate with different versions of the software in different countries or for different subsidiaries or divisions and you need to understand the differences when planning for a new project
- You need to compare a snapshot of the application from the past with the current version in order to track down system changes that are causing problems
- A slightly different situation is when you have a packaged application for which you have made customizations and the vendor delivers a new release and you need to assess the impact of the new release on your customizations.

What information is needed and why

Performing an analysis for any of these circumstances involves comparing a large set of system and source code information. At a high level you might investigate some of these types of information for differences between the versions:

- Commands
  - Parameters
  - Command processing and validity checking programs called
- Database
  - Fields
  - Keys
  - Relationships
  - Logical files over each physical file
  - Constraints
  - Triggers
  - Select/omit criteria
- Programs
  - Business rules

    - Bound modules
    - Program references
    - Subroutines
    - Bound service programs
    - Procedures
    - SQL queries
- Source code
    - Individual statements added, changed or deleted

Obviously this is a lot of information and accomplishing this task involves these primary capabilities:

- Collecting and storing this information for two versions
- For the fourth case listed in the top section you actually need to triangulate between three versions of the source and objects: the original base package, your customizations, and the new version of the base package
- Analyzing the data and reporting on the differences.

Databaorough's X-Audit product provides the functionality to do these kinds of analysis.

Steve Kilner

© Databorough