

# The Smoking Gun

## *Finding the ROI in Large Software Development Projects with Software Engineering Solutions*

By Greg Spehar ([spehar@full-knowledge.com](mailto:spehar@full-knowledge.com))

### Executive Summary

Finding and removing software defects and security vulnerabilities has a direct impact on the cost of software development projects. The National Institute of Standards and Technology (NIST) reports that "...80% of development costs [are spent] on identifying and correcting defects." The NIST continues to conclude that "...all errors cannot be removed, more than a third of these costs, or an estimated \$22.2 billion, could be eliminated by an improved testing infrastructure that allows earlier and more effective identification and removal of software defects." (NIST June 28,2002)

The following discussion will focus on those frameworks and processes that will help in finding bugs before the test process through the use static analysis. Most developers have been using standard static analysis programs for defect detection for years. With latest Static Analysis tools combined improvements in the static analysis engines and introduction of metric and architecture management the potential return on investment (ROI) becomes ever more promising. In addition, with the use of Internet web technologies these results can be defined and distributed immediately, reducing the time required to manage the results generated by the Static Analysis tools.

This document describes the evaluation of a Static Analysis solution from three different perspectives to understand potential ROI and to assist in identifying the "smoking gun" or proof required to justify this type of tool within the target organization. The ROI calculation is based on three different perspectives:

- Defect Management – Automated discovery of code problems before test such as memory leaks, null pointers and security vulnerabilities.
- Architecture Management – Automated discovery and enforcement of architectural rules such as API and package enforcement.
- Metrics Management – Automated discovery and enforcement of metrics such as code complexity, churn, and defect density.

There are two primary process changes that need to occur to take advantage of this new technology:

- Moving from the detection of errors to the prevention of errors.
- Moving from manual processes to automated solutions before the testing stage.

#### Target Companies:

- IT Organizations
- Software Companies
- Embedded Systems

#### Target Audience:

- Executives
- Directors
- Software Managers

#### Our Goal:

Creating wealth by assisting in delivering your software projects to market fast and reliably.

#### Our Guarantee:

We guarantee that we will create wealth with a ROI that will exceed 1000%, or the services will be one third (1/3) the price.\*\*

\*\* Only applies to the wealth creation package.

#### Contact:

Full Knowledge LLC  
2477 Santa Rita Rd #32  
Pleasanton, CA, 94566

Office: 925-484-8490  
Cell: 503-332-3663

Email:  
[sales@full-knowledge.com](mailto:sales@full-knowledge.com)



## Process Change - Detection vs. Prevention

In the quest to develop and maintain ever growing code bases, the need to have tools that can automatically identify issues or concerns will grow. If current bug rates of several bugs per thousand lines of code continues to be an industry standard, the amount of bugs per system will continue to grow to the point that very large systems will require such extensive testing and feedback mechanisms that these programs will not be maintainable or will only be maintainable by very large companies with accessible resources.

This industry norm is currently focused on the “Defect Detection” strategy. With companies having extensive investments in the testing process or in the bug correction process. These companies make investments to the level where customers and testing resources have to expend tremendous effort by participating in the quality correcting mechanism of bug identification and resolution.

The new strategy that is becoming important is the “Defect Prevention” strategy. This strategy has been embraced by many companies in the form of more manual processes within the development process and more complete and specific business specifications with the associated technical specifications. Even then organizations can expend many more resources and effort to ensure the technical design is correct and error free.

## Automated Prevention

The ability to run incremental tests prior to submission of code to the code tree or the submission of an entire code base to testing is the next technological advantage. This type of testing is defined within the industry as “Static Analysis”. With the current static analysis tools and technologies, the written code or code base can be compiled and rules engines can be defined to simulate an execution to find critical bugs that would otherwise be found in the test process or in the field. With the new Static Analysis Tools, the noise that has detracted from the value of these tools in the past has been reduced such that these tools are currently very cost effective. In addition, combining static analysis with metric and architecture data further increases the accuracy of critical errors.

These mechanisms can be automated and managed to ensure that the users of the tools will prevent critical logic errors (bugs) from entering the code tree or the testing process. This process of prevention will change the industry in two ways:

- Testing will be able to focus on performance and conformance to business specs.
- Customers will not have to endure the problem of failed applications in the field.



## Business Impact

The impact of this process change will positively change organizational business objectives such as the need to control software defects, to identify security vulnerabilities, to identify and manage the software metrics and KPI, to ensure offshoring success and quality, to prevent “Fire Drills” at the time of deployment, to allow surgical testing and to allow code management mobility.

A detailed discussion on how each of these business areas positively impact the organization can be found in the document “Finding the ROI in Business Objectives”. Where the following sections will focus on how to identify the “Smoking Gun” within the Defect, Architecture and Metric Management processes such that an Opportunity ROI can be generated from the discovered “Smoking Gun” data.

### **Finding the Smoking Gun - Defect Management**

#### The Goal

This section provides a framework for a company to find those specific bugs, security vulnerabilities or issues in software development that reduce test effort and increase a product’s reliability in the field. This translates to real economic value. Bugs found prior to traditional testing or found prior to a product’s entry into the field reduce development costs considerably as mentioned by the NIST.

#### Setting Up Initial Test

When evaluating the Defect Management process it is important to understand that two methods exist for capturing these defects, developer based tools and system/server based integration tools. Both can provide a level of feedback that can prevent errors from entering test or the field. The most effective Defect Management tools will be developer tools that report status at the moment a code change occurs. The second method is to find errors at the system level or integration point. This type of analysis can review the entire code base and accumulate all issues into one concise location. This second method is what is required for finding the “Smoking Gun”.

Since there must be a proof that these issues can be caught prior to entering the test process or deployment to the field, a comparison of older versions of the code, that have already passed through test, must be reanalyzed with the Static Analysis tools. The results from the Static Analysis tools will be compared to the actual issues found in test and the field.



## Gathering Historical Data

Finding historical data can be a problem for some organizations. But most companies have bug-reporting systems that are used extensively by the testing organization. In addition, other mechanisms such as email or spreadsheets may be sources that have captured software failures from test and from the field. The following are the steps for preparing the data:

- 1) Identify a version of the software that has been tested and released to the field
- 2) Gather all reported application failures and their resolutions
- 3) Translate or enter the file name and method name of the file impacted or changed
- 4) Enter the range of line numbers impacted by the change
- 5) Enter this information into a spreadsheet format
- 6) Find all critical issues, a set of 50-100 should be adequate

## Finding the Smoking Gun

Run the Static Analysis tool called , a system level tool that runs a set of rules against all the code that is to be analyzed, on the same release as analysed previously. Use the resulting data set to find matching issues with the previous analysis. The resulting matches are the “Smoking Gun” results. See the Appendix for detailed example calculations for an Opportunity ROI using this information.

## Finding the Smoking Gun - Architecture Management

### The Goal

This section provides a framework for a company to better manage the logical concerns of the software that needs to be maintained over time. The quantifiable effects of software that has a consistent architecture fall into several categories, they include:


- 1) Knowledge of Application (Reduce costs and reduce injection of bugs)
- 2) Testing Requirements (Isolating components reduces testing requirements.)
- 3) Management of Resources (Isolating code increases flexible management.)

Resulting Architecture Opportunities

- a. Benefits for Code Management Mobility
- b. Benefits for Surgical Testing

### Setting Up Initial Test

When evaluating the Architecture Management process it is important to understand that a tool is required to understand the relationships in a clear and consistent manner. This tool will be required to view and change relationships between files to ensure that the required relationships can be properly managed to secure the logical concerns that are partitioned in a rational manner. These



decisions need to then be distributed to those that make changes such that the decisions are then enforced.

The architecture can be quantified through several assumptions, they include:

- 1) The amount of time to learn new code
- 2) The amount of time to test an application
- 3) The amount of time to relearn existing code
- 4) The opportunity cost to develop the software

In the effort to find the “Smoking Gun”, there must be proof that these issues can have a concrete impact to development efforts without extensive investment in behavioural testing. To remove this issue of behavioural testing, the opportunity costs associated with the time to test developed software will be used as the comparison. The required analysis will include a comparison of older versions of software components that have already passed through testing **without** issues compared to those that passed through testing **with** multiple issues.

### Gathering Historical Data

Defining this relationship between those changes that were less problematic versus those that are more problematic should represent an indication of complexity. Code that is well understood and conceptually partitioned properly should, by definition, result in fewer issues when faced with code changes and the resulting test requirements. The defining assumption within this strategy is: *The changes to the code base occurred with a resources of equal competence.*

The resulting analysis should provide information such as 1 day of testing versus 5 days of testing or 1 week of testing vs 2 weeks of testing. The approach taken to gather the data includes:

- 1) Identify a version of the software that has been tested and released to the field
- 2) Gather all reported application failures
- 3) Enter the file name and method name of the file impacted or changed
- 4) Gather all names of all files and all changes for the release
- 5) Group files with changes and reported errors
- 6) Group files by component or subsystem

### Finding the Smoking Gun

Run the Static Analysis InSight Architect tool, a system level tool that diagrams all the relationships between files, on the same release as analysed above. Record the relationships into components and out of components as another factor. The “Smoking Gun” will be a correlation between errors found related to the amount of coupling, or interaction, between systems. The more self-contained a component the less likely a bug or testing issue would arise. The more dependencies a component has on other components the more likely an error will occur. See the Appendix for detailed example calculations for an Opportunity ROI using this information.

## Finding the Smoking Gun - Metrics Management

### The Goal

This section provides a framework for a company to assist in creating a mechanism to manage the metrics of the software system to predict when a software bug will be most likely to occur. Identification of these metrics can translate to bottom line value since most customers have quantifiable data for bugs that reach test or the field.

### Setting Up Initial Test

When evaluating the Metric Management process it is important to understand that a tool is required to create and manage the software system metrics build over build. This tool will be required to view and track the metrics of the software over time. This information must be used to make decisions that would prevent the injection of defects into the system.

Management of the coding process through metrics makes several assumptions:

- 1) Metrics evaluation of errors can predict future errors
- 2) Metrics gathering is part of the process (With a very small overhead for collection)
- 3) Metrics can be gathered and acted on before being obsolete

Since there must be proof that metrics can have a concrete impact to development efforts without extensive investment in behavioural testing, the opportunity costs associated with the prevention of field defects will be the focus. The required analysis will include a comparison of older versions and software components that have already passed through testing **with** issues compared to the metrics generated for the code containing the errors.

### Gathering Historical Data

Defining this relationship between the errors and the metrics of the code being changed should produce a profile of what code will be more prone to generate an error when the code is changed. Code that is well understood and conceptually partitioned should by definition have consistent well-defined metrics and therefore should result in fewer bugs when faced with code changes. The approach taken to gather the data includes:

- 1) Identify a version of the software that has been tested and released to the field
- 2) Gather all reported application failures
- 3) Enter the file name and method name of the file impacted or changed
- 4) Gather all names of all files and all changes for the release
- 5) Group files with changes and reported errors
- 6) Group files by component or subsystem



## Finding the Smoking Gun

Run the Static Analysis Metrics profile, gathering all the metrics data from the release as analysed above. Record the metrics such as method and file level McCabe complexity, number of returns, number of loops and amount of code. The “Smoking Gun” will be a correlation between errors found related to the metrics profile defined.

The last test is to take a follow-on release that has been tested and has been released to the field and run the same analysis to see if the cross section of metrics to reported citings WOULD HAVE prevented defects found in test and also found in the field.

### Example Calculation

Since these three processes, Defect, Architecture and Metric, can define the overall value (ROI) in using these types of Static Analysis tools, a general framework has been presented that will assist in the process of proving or finding the “Smoking Gun”. The final component is to work through an example ROI that would allow the creation of a model specific to an organization to further define the Opportunity Costs and outline a execution timeline that provides potential project windows.

The Appendix will define that model and an example calculation. The final section will outline a simple timeframe that can be used within a planning and cost assessment.

The following are the sections within the Appendix:

- A. ROI – Calculating Assumptions
- B. ROI – Delivered Products
- C. ROI – Sample Calculation
- D. Planning - Corporate Portfolio Rollout

## Appendix A: ROI - Calculating Assumptions

When calculating a complete ROI for a project each project will have a unique set of requirements. This section will define several assumptions such that each project can be assessed based on specific parameters. The calculation will have three ROI components:

- 1) Quality Based ROI – Prevention of bugs into the test process and into the field
- 2) Architecture Based ROI – Opportunity costs of component architectures errors
- 3) Metric Based ROI – Opportunity costs of errors based on predictive metrics

### Quality Based ROI Assumptions

Based on the “bugs” are considered to be of critical nature there will be an entire range of bugs that are not “critical” but “important” or “must do” or “need to fix”. These are all the additional “bugs” that can be found by the Static Analysis tool but are not determined to be a “Smoking Gun”. The calculation assumptions are:

- 1) Calculation of man-hours saved through automating a code walk-through
  - a. Assumed cost savings \$500 per hour
  - b. Review for 1 hour 500-1000 LOC covering 3 issues
- 2) Calculation of defects caught before entering testing
  - a. Assumed cost per defect \$500
- 3) Calculation of defects caught before entering the field
  - a. Assumed cost per defect \$10,000 (Capers uses \$15,000)

Source: Applied Software Measurement – Capers Jones, 1996

### Architecture Based ROI Assumptions

Based on the components that were considered to be part of the system, categorize the components into “component” and “dependent component”. This classification can provide an analysis that includes the Opportunity Cost to test and to replace components. The calculation assumptions are:

- 1) Calculation of Code Management Mobility
  - a. Manpower per Component 400KLOC/20KLOC – 5 person/20KLOC
  - b. Assume coding resource is \$100K/year on high end - \$50K/year on low end
- 2) Calculation of Surgical Testing
  - a. Assumed cost savings 5 testing days vs 1 testing day (\$50K per day)
  - b. Assume testing window is 2 months (Compare 40 Tests to 8 Tests)

### Metric Based ROI Assumptions

Based on the metrics that were considered to be part of the defect analysis, models can be defined to classify code that may generate errors. This classification can provide an analysis that includes the Opportunity Cost to test and to replace components. The calculation assumptions are:

- 1) Calculation of Metrics on Errors Before:
  - a. Testing – 5% of pre-testing errors
  - b. Release to the field - 20% of pre-field errors
- 2) Calculation based on Code Churn Rate of 20% per year (Amount of code changes)

## Appendix B: ROI – Delivered Products

When reporting the ROI for a project there are three parts that need to be performed. The first is to identify the potential Defect ROI on the system. Then the Architecture ROI will determine the real value of the product over time. The final ROI calculation is the Metric ROI for tracking projects over time. The combination of these three processes will assist in producing quantifiable data that can measure the value of the Static Analysis tools from developer to integration into the build process.

### Part 1 – Defect ROI

The potential Defect ROI will determine the possible return on a small project. This calculation will provide enough data to ensure that a return can be gained through the use of the Static Analysis tools. The deliverables for this phase is the following:

ROI Calculation for Defect Management

- 1) ROI document that describes the investment opportunity
- 2) Documentation, Training Materials, All Citing Reports, Architecture Report, Static Analysis Report, Integration Guide, Technical Guide and General Recommendations
- 3) A Defect ROI based calculation is provided as an example in the Appendix - Sample ROI Calculation:
  - Phase 1 Defect ROI

### Part 2 – Architecture ROI

The potential Architecture ROI will determine the possible return on a *large* project. This calculation will provide enough data to ensure that a return can be gained through the use of the Static Analysis Tools. The deliverables for this phase include:

ROI Calculation for Architecture Management

- 1) Documentation, Training Materials, Architecture Report, Architecture Guide, Architecture Actions, Architecture Configuration Files, Enforcement for Desktop-Server Static Analysis Guide and General Recommendations.
- 2) An Architecture ROI based calculation is provided as an example in the Appendix - Sample ROI Calculation:
  - Phase 2 Architecture ROI

### Part 3 – Metric ROI

The potential Architecture ROI will determine the possible return on a *large* project. The deliverables for this phase include:

ROI Calculation for Metric Management

- 1) Documentation, Training Materials, Metrics Report, Metrics Guide, Metrics Actions, Metric Configuration Files, Enforcement for Desktop-Server Static Analysis Guide and General Recommendations

- 2) A Defect ROI based calculation is provided as an example in the Appendix -  
Sample ROI Calculation:
- Phase 3 Metric ROI

### Appendix C: Sample ROI Calculation

The following are three sample calculations of the ROI based on “Opportunity Costs”. This example is intended to provide a framework for real world software systems.

#### Example – Part 1 Defect ROI

As an example lets take a look at corporate project that is considered to be large. The entire company might have 10 Million Lines of Code (MLOC). There are 5 products that are delivered within this company each consisting of 2 MLOC (5 products at 2 MLOC). The following calculation can be defined:

- 1) Quality Control Opportunity Costs
  - a. Automated Walk-through
    - i.  $2 \text{ MLOC} * \$500 \text{ per } 1000 \text{ LOC} = 2000 * \$500 = \$1\text{M}$
  - b. Prior to Testing Caught Defects
    - i.  $2 \text{ MLOC} (1 \text{ defect/ } K\text{LOC}) * \$500 \text{ per defect} = 2000 * \$500 = \$1\text{M}$
  - c. Prior to Field Caught Defects
    - i.  $2 \text{ MLOC} (1 \text{ defect/ } 10 \text{ KLOC}) * \$15\text{K per defect} = 200 * \$15\text{K} = \$3\text{M}$
  - d. Total Potential Value
    - i.  $\$1\text{M} + \$1\text{M} + \$3\text{M} = \$5 \text{ Million}$
- 2) Product Costs Plus Manpower Investment
  - a. Software and Manpower (Calculations will vary)
    - i. Build and System Level = \$100K (SW ~ \$50K)
    - ii. Architecture Level = \$100K (SW ~ \$15K)
    - iii. Developer Level = \$300K (SW ~\$35K at 20 developers)
    - iv. Total Investment = \$500K
  - b. Fixed Assets
    - i. Facilities – Not Included (Assumed part of Manpower costs)
    - ii. Hardware – Not Included (Hardware Costs Minimal)
- 3) Deliverables and Calculated ROI
  - a.  $\text{Opportunity Costs} / \text{Investment} = \$5\text{M} / \$0.5\text{M} * 100\% = 1000\% \text{ ROI}$
  - b. Inputs – The code (2MLOC), the build scripts, a server and manpower
  - c. Outputs – Documentation, Training Materials, All Citing Reports, Architecture Report, Static Analysis Report, Integration Guide, Technical Guide and General Moving forward recommendations.

**Example – Part 2 Architecture ROI**

As part 2 is considered the overall benefit will be the same. The added capabilities will also include the capacity to better manage the architecture. In this case the calculations will consider the opportunity costs of NOT managing the architecture. As before the focus will be on one 2 MLOC application. We will assume that this application will have 5 primary components. The management or non-management of the components will then be calculated. The following calculation can be defined:

- 1) Quality Control Opportunity Costs (As previously):
  - a) Opportunity Costs = \$5 Million
  - b) Total Investment = \$500K
- 2) Architecture Control Opportunity Costs
  - a) Code Management Mobility (Assume 20KLOC per Developer)
    - i) Clear Separation of Components (No Coupling)
      - (1) Manpower per Component 400KLOC/20KLOC = 20 People
      - (2) Total Manpower for Product is 5 \* 20 people = 100 People
      - (3) Two Components Managed by High Tier Development
        - (a) Two (2) \* 20 people at \$100K year = \$4M / Year
      - (4) Three Components Managed by Lower Tier Development
        - (a) Three (3) \* 20 people at \$50K year = \$3M / Year
      - (5) Total Management Costs
        - (a) \$4M / Year + \$3M / Year = \$7M / Year
    - ii) Coupling of Component Requires Single Integrated Team
      - (1) Assume costs as mentioned above
      - (2) Five Components Managed by High Tier Development
        - (a) Five (5) \* 20 people at \$100K year = \$10 M / Year
    - iii) Total Benefit for having Code Management Mobility
      - (1) \$10 M - \$7 M = \$3 M
      - (2) Or a 30% improvement in manpower maintenance costs
  - b) Surgical Testing
    - i) Cost Assumptions
      - (1) Testing Cost per Component 400KLOC = 1 Day
      - (2) Testing Window Cost
        - (a) Cost for 1 Day of Testing = \$50K
          - (i) Includes Manpower and Equipment and Support Software
        - (b) Cost for 1 Week of Testing = \$50 \* 5 Days = \$250K
        - (c) Standard Testing Window is 2 Months or 8 Weeks
        - (d) Cost for a Testing Window = 8 \* \$250K = \$2M
      - (3) Assume Testing Window is Fixed (No Gating Bugs)
        - (a) No Coupling of Components = 40 Individual Rounds of Tests
          - (i) Assuming Regression and Components can be tested individually in one day
        - (b) Coupling of Components = 8 Individual Rounds of Tests
          - (i) Assuming Regression and NO Components tested in one day
      - (4) Testing Captures Percentage of Bugs
        - (a) Assume Removal Rate is 20% and Removal Occurs at each Test
      - (5) Bugs that Escape Testing
        - (a) Percentage of Critical Bugs assumed to be 5%
        - (b) Cost of Critical Bugs to Field are \$10K

- ii) Resulting Opportunity Cost (See Testing Iteration Cost Matrix Table)
  - (1) System Coupled Testing Cost of Escaped Bugs
    - (a) 8 Rounds of Testing - \$8.4M
    - (b) 40 Rounds of Testing - ~\$5K (a Difference of \$8.4M)
  - (2) Component Coupled Testing Cost of Escaped Bugs
    - (a) 8 Rounds of Testing - \$1.7 M
    - (b) 40 Rounds of Testing - ~\$1.3K (a Difference of \$1.7 M)

**Example Phase 2 ROI Table #1: Testing Iteration Cost Matrix Table**

Testing Iteration Cost Matrix				
Total LOC	2,000,000			
Components	5			
Removal Rate	20%			
Error Rate	5%			
Percent Critical	5%			
Field Costs	\$10,000.00			
	The System	Field Cost	A Component	Field Cost
Test Iteration	100,000	NA	20000	NA
1	80,000	NA	16,000	NA
2	64,000	NA	12,800	NA
3	51,200	NA	10,240	NA
4	40,960	NA	8,192	NA
5	32,768	NA	6,554	NA
6	26,214	NA	5,243	NA
7	20,972	NA	4,194	NA
8	16,777	\$8,388,608	3,355	\$1,677,722
9	13,422	\$6,710,886	2,684	\$1,342,177
10	10,737	\$5,368,709	2,147	\$1,073,742
11	8,590	\$4,294,967	1,718	\$858,993
12	6,872	\$3,435,974	1,374	\$687,195
13	5,498	\$2,748,779	1,100	\$549,756
14	4,398	\$2,199,023	880	\$439,805
15	3,518	\$1,759,219	704	\$351,844
16	2,815	\$1,407,375	563	\$281,475
17	2,252	\$1,125,900	450	\$225,180
18	1,801	\$900,720	360	\$180,144
19	1,441	\$720,576	288	\$144,115
20	1,153	\$576,461	231	\$115,292
21	922	\$461,169	184	\$92,234
22	738	\$368,935	148	\$73,787
23	590	\$295,148	118	\$59,030
24	472	\$236,118	94	\$47,224
25	378	\$188,895	76	\$37,779
26	302	\$151,116	60	\$30,223
27	242	\$120,893	48	\$24,179
28	193	\$96,714	39	\$19,343
29	155	\$77,371	31	\$15,474
30	124	\$61,897	25	\$12,379
31	99	\$49,518	20	\$9,904
32	79	\$39,614	16	\$7,923
33	63	\$31,691	13	\$6,338
34	51	\$25,353	10	\$5,071
35	41	\$20,282	8	\$4,056
36	32	\$16,226	6	\$3,245
37	26	\$12,981	5	\$2,596
38	21	\$10,385	4	\$2,077
39	17	\$8,308	3	\$1,662
40	13	\$6,646	3	\$1,329

So in concluding the value in part 2 we can see that the overall investment in time and ROI could come to the following:

- 1) Architecture Control Investment costs for 2MLOC
  - a. Mobility and Surgical (5 Man Years) = \$500K
- 2) Deliverables and Calculated ROI
  - a. Opportunity Costs / Investment = \$3 M + \$8.4 M / \$500K \* 100 Percent  
= \$11.4M / \$0.5M \* 100% = 2280% ROI
  - b. Inputs – The code (2MLOC), the build scripts, a server and manpower
  - c. Outputs – Documentation, Training Materials, Architecture Report, Architecture Guide, Architecture Actions, Architecture Configuration Files, Enforcement for Desktop-Server Static Analysis Guide and General Recommendations.

**Example – Part 3 Metric ROI**

As phase 3 is considered the overall benefit will be the same from the previous phases. The difference is the other components will be added to the project. In this final phase the additional burden will include the Metric Management of the code base. In this case the introduction of the metrics management will allow a company as a systems provider to better understand an issue prior to its occurrence. As before the focus will be on one 2 MLOC application. We will assume that each of the 5 applications will have 200 files/classes each with 20 main functions/methods per file/class. The management or non-management of the components can then be calculated. The following are the calculations:

- 1) Quality Control Opportunity Costs (As previously for 2MLOC):
  - a) Opportunity Costs = \$5 Million
  - b) Total Quality Investment = \$500K
- 2) Architecture Control Opportunity Costs (As Previously for 2 MLOC)
  - a) Opportunity Costs = \$8.4 Million + \$3 Million = \$11.4 Million
  - b) Total Architecture Investment = \$500K
- 3) Metric Control Opportunity Costs for 2 MLOC
  - a) Management Entities
    - i) Applications 5
    - ii) Files/Classes = 5 \* 200 Files per app = 1000
    - iii) Functions/methods = 1000 \* 20 Func/Meth = 20000
  - b) Management Measurement
    - i) Complexity
    - ii) Citings
    - iii) Reported
      - (1) True Citings
      - (2) Test Defects
      - (3) Field Defects
    - iv) Lines of Code
    - v) Maintainability
    - vi) Churn
  - c) Breakdown of LOC
    - i) Application Components – 2MLOC / 5 Applications = 400KLOC per app
    - ii) Files/Classes – 400KLOC / 200 Files/Classes = 2KLOC per File/Class
    - iii) Functions/Methods – 2KLOC / 20 Func/Meth = 100LOC per Func/Meth
  - d) Typical Error Breakdown
    - i) Assume 20% Churn and 5% Initially before Testing
      - (1) Application Components Errors – 400KLOC \* 20% \* 5% = 4K Errors
      - (2) Files/Classes – 2KLOC \* 20% \* 5% = 20 Errors
      - (3) Func/Meth = 100LOC \* 20% \* 5% = 1 Errors
    - ii) Potential Critical Bugs Escape assume 20% of Errors
      - (1) App. Components Errors – 20K Errors \* 20% \* 20% = 800 Critical Errors
      - (2) Files/Classes – 100Error \* 20% \* 20% = 4 Critical Errors
      - (3) Func/Meth – 5 Errors \* 20% \* 20% = 0.2 Critical Errors
  - e) Control Chart Management of Code Base
    - i) Application Components Control Charts = 5 Charts
    - ii) Files/Classes Control Charts = 1,000 Charts
    - iii) Func/Meth Control Charts = 20,000 Charts
  - f) Management of Control Charts



- i) Continued Control Management of 5 applications
- ii) Yellow and Red Indication (Stop light management)
  - (1) Control Chart only visible if in Red or Yellow Zone
    - (a) Files/Classes
    - (b) Func/Meth
  - (2) Build over build management of indicators
- g) Resulting Benefits of Control Chart Management
  - i) Quality Control of Applications to Function Level
  - ii) Reduction of Critical Errors Near Zero
  - iii) Assumed Critical Error costs of \$10,000
  - iv) Opportunity Cost Calculation
    - (1) 2 MLOC System = 5 Components \* 4K Errors \* \$500 = \$20 Million
    - (2) 2 MLOC System = 5 Components \* 800 Errors \* \$10K = \$40 Million

So in concluding the Value at Phase 3 we can see that the overall investment in time and ROI could come to the following:

- 1) Metric Control Investment costs for 2MLOC
  - a) Metric Control Effort (5 Man Years) = \$500K
  - b) Metric Control Software and Equipment = \$500K
- 2) Deliverables and Calculated ROI
  - a) Opportunity Costs / Investment = \$60M / \$1M \* 100% = 6,000% ROI
  - b) Inputs – The code (2MLOC), the build scripts, a server and manpower
  - c) Outputs – Documentation, Training Materials, Metrics Report, Metrics Guide, Metrics Actions, Metric Configuration Files, Enforcement for Desktop-Server Static Analysis Guide and General Recommendations.

## Totals for All Applications for 10MLOC

The totals for the entire set of 5 Applications:

- 1) Quality Control – Defect ROI
  - a) Investment Costs for 10MLOC – 5 \* \$500K = \$2.5 Million
  - b) Opportunity Costs for 10MLOC – 5 \* \$5M = \$25 Million
- 2) Architecture Control - Architecture ROI
  - a) Investment Costs for 10MLOC – 5 \* \$500K = \$2.5 Million
  - b) Opportunity Costs for 10MLOC – 5 \* \$11.4M = \$57 Million
- 3) Metric Control - Metric ROI
  - a) Investment Costs for 10MLOC – 5 \* \$1M = \$5 Million
  - b) Opportunity Costs for 10MLOC – 5 \* \$60M = \$300 Million
- 4) Total ROI
  - a) Investment Total = \$10 Million
  - b) Opportunity Costs Total = \$382 Million
  - c) ROI = 3820%

*NOTE: These calculations are intended to supply an example model for thinking about the potential opportunity for changing software management mechanisms from a defect detection strategy to a defect prevention strategy. Individual companies and programs will differ in these assumptions. In addition, this analysis does not calculate the manpower savings and real world market timing advantages that may occur. This analysis assumes the cost structures do not change; so additional calculations can be assessed that would indicate bottom line cost realization.*

## Appendix D: Planning - Corporate Portfolio Rollout

When implementing this change into an organization there will be four primary phases that will be required. The first phase will focus on the proof of the Defect ROI Value. The second phase, based on the success of the first, is the proof of the Architecture ROI and Rollout of the Defect Management. The third phase is the proof of the Metric ROI and Rollout of Architecture Management. The fourth and final phase is the Rollout of the Metric Management. The following is to provide an example of a typical rollout to a software organization with 5 product lines.

### Phase 1 – Defect ROI Proof (1-3 Months)

*Defect Management:*

- *Initial project identification and initial Rollout to project#1*
- *Actual integration work: 2 Weeks*

Note: Additional projects may be integrated into Defect Management at Phase 1.

### Phase 2 – Architecture ROI Proof and Defect Rollout (3-6 Months)

*Architecture Management:*

- *Initial Rollout to project#1*
- *Actual integration work: 2 Weeks*

*Defect Management:*

- *Projects #2, #3, #4, #5: Total time is 2 weeks x 4 Projects = 8 weeks*

Note: Additional projects may be integrated into Architecture Management at Phase 2.

### Phase 3 – Metric ROI Proof and Architecture Rollout (3-6 Months)

*Metrics Management:*

- *Initial Rollout to project#1*
- *Actual integration work: 2 Weeks*

*Architecture Management:*

- *Projects #2, #3, #4, #5: Total time is 2 weeks x 4 Projects = 8 weeks*

Note: Additional projects may be integrated into Metrics Management at Phase 3.

### Phase 4 – Metric Rollout (2-3 Months)

*Metrics Management:*

- *Projects #2, #3, #4, #5: Total time is 2 weeks x 4 Projects = 8 weeks*