krugle

White Paper

The 4 Hidden Costs of Code Maintenance

Summary

Code maintenance has grown to become the largest budget item in most software development organizations [1][2]. Many significant cost and quality problems spawn and proliferate in common code maintenance development activities. This paper examines the source of these problems and explains how modern code search technology is being used to remedy these issues.

This paper addresses the most common and costly code maintenance issues and solutions. The key code maintenance issues are:

1) Steep learning curve (faced by developers working with someone else's code)
2) Insufficient research and planning
3) Duplication of effort
4) Poor compliance with coding standards and practices

Enterprise-wide code search and analysis is a technology which can be used by development organizations to drive down maintenance costs, defects and waste associated with these problems. By breaking down information access barriers to code, this technology can save organizations millions of wasted dollars each year and help refocus development on strategically significant opportunities.

A look at the problem

Today's software development methodologies and tools do a good job of making major software releases more manageable. Costs, quality and time-to-market are reasonably predictable and well understood.

Major releases get the best resources, planning and support.

The challenge involves work that is done between major releases. This work is often referred to as "evolutionary development" or "code maintenance."

The problem: development performed between releases is prone to suboptimal resourcing, limited planning and poor design cooperation.

As a result maintenance generates more than its fair share of cost and quality problems. The core problems stem from common development practices and situations:

- Developers frequently must work with code they didn't write and don't understand
- Complex component and distributed architectures make learning unfamiliar code extremely difficult
- The inability to access and share relevant insight (code, developer contact, pertinent documentation) is ubiquitous

- Developers lack the time or tools to assess the impact of code change and determine change requirements across the organization
- Mergers and acquisitions, multisite development teams, laptops, and management autonomy result in little or no ability to detect or prevent the same problem from being “fixed” independently and differently across projects
- The envelope of time pressure around the preceding issues results in incomplete fixes, “hacks”, and code entropy.

How big of a problem is code maintenance?

In companies with 100+ developers and/or an active codebase of 500K+ lines of code, software maintenance will account for more than half of the overall development budget. The problem is greater in larger organizations. The result is that code maintenance increasingly chews up a greater percentage of IT spending, stealing budget from projects that are needed to drive future financial returns.

For development organizations, this problem is growing. The maintenance liability will continue to grow because of three factors:

- Code interdependencies are increasing
- The rate of code change is increasing
- Poor code maintenance spawns additional code maintenance issues.

This is a significant problem with aggressive growth. It is significant and expensive cancer that threatens virtually all software development organizations. Although organizations are well aware of the symptoms, most do not understand the common driving forces or practical solutions.

The Evolving Landscape of Software Development

	Past	Present
Release cycles	Long	Short
Requirements	Detailed	Basic
Interdependencies	Few	Many
Stack change rate	Quarterly/Yearly	Daily/Weekly
Planning	Full	Ad-hoc
Points of coordination	Few	Many
% Developer time on someone else's code	<25%	>75%
Maintenance as % of total effort	Less than 50%	More than 50%
Ability to control lifecycle quality	High	**??**
Ability to control lifecycle costs	High	**??**
Ability to plan and manage dev spending	High	??

Table 1. Key Cost and Quality Drivers Software Development

In the past, code was developed around a relatively static and predictable framework. Applications relied on simple component architectures that were stable and predictable. As a result, most development was applied to major releases that could be carefully managed against a modest number of predictable dependencies. Tools and methodologies (e.g., waterfall, xtreme) ensured the efficient creation of these monolithic applications through finely detailed requirements, development and testing planning.

The current development landscape is very different (see Table 1).

Code today is more complex, more dynamic and built from component modules for – at least in theory – greater development leverage and improved quality. Many independent teams are involved in the varied component modules that create a working stack of software. Development cycles for these

components are short. The overall rate of change within a complete application code stack is very high. As a result, the frequency of "post release" development issues (defects, security vulnerabilities, performance issues, compatibility issues) has increased.

This accelerating cycle of change forces development behaviors out of the carefully managed and predictable envelope of ALM best practices (detailed analysis, requirements definition, test planning, etc) and into the realm of "no time to do it right; we'll do the best we can now and patch it later after it's live."

Less experienced programmers and outsourced development teams are typically assigned to perform maintenance or evolutionary development. Less experienced staffers often lack deep code knowledge. They have limited visibility into relevant coding examples, dependences or documentation. Much of their work is left to their discretion (uncontrolled or minimally controlled processes) and they are forced to work under tight deadlines. The vast storehouse of reusable components goes mostly unused.

Rapidly changing code components combined with time pressure, lack of experience and poor coordination result in costly, chronic problems in software maintenance. As a result, maintenance - and evolution-driven code changes are frequently the source of costly (but avoidable) problems.

The 4 Critical Issues in Code Maintenance

Issue 1 **Steep learning curve**

- In most cases, the person who works on a maintenance issue isn't the original code author
- Maintenance developers tend to be newer or less experienced developers
- Code (inter) dependencies make learning very difficult to understand, particularly if the code is external to the known project and multiple versions of the code are involved – this is when the right code can't be found or easily inspected

As a result, maintenance code changes are commonly performed with an incomplete understanding of the code. Important nuances or edge cases may be overlooked or mishandled due to the fact that code cannot be adequately learned within the time allotted for the maintenance work.

Issue 2 **Insufficient research and planning**

- Once a release is in the maintenance phase of the lifecycle, there is significant pressure to remedy the problem quickly.
- Despite best intentions and heroic efforts, many changes implemented during maintenance are not carefully planned or executed.
- Time to research existing fixes for similar problems is seriously limited.
- Low visibility into the collective body of relevant work – for similar fixes, lessons learned, testing resources, etc. This information is typically stored in systems that the maintenance developer doesn't have access to or knowledge of.

- It is impossible to understand the current and downstream impact of proposed changes to other projects that use or depend on the code being changed. Performing thorough impact analysis is desirable, but not practical due to development system information access barriers.

Without proper scoping and coordination, changes are frequently incomplete and result in greater than normal level of downstream defects. Poor maintenance practices create additional maintenance issues.

Issue 3 **Duplicated coding**

- For any developer involved in post release code changes, visibility into the body of code changes that are "in development" and "successfully completed" is extremely limited.
- Manually managed code libraries, practice centers, code reviews and collaboration tools are ineffective knowledge transfer mechanisms. This is because developers are coders not catalogers, communicators or documenters – particularly with time sensitive or complex issues.
- Faced with little or no relevant information or guidance, developers will devise and implement their own changes. In almost all cases, the change for a given issue will vary widely from developer to developer.

The presence of the same issue in different bodies of code will result in multiple "fixes" to address the same issue. Not only does the planning, coding and testing of more than one fix for the same problem represent wasted effort, but the downstream maintenance and support liability created by each "fix stream" is completely avoidable.

Issue 4 **Divergence from coding standards**

The pressure for timely fixes, developer unfamiliarity with the code and less-than-rigorous planning are common factors in software maintenance. These common factors result in uneven coding practices such as the following:

- Modifying local instances of code components without merging changes into the main branch.
- Usage of components or code that is unauthorized (for technical or legal reasons).
- Usage of coding techniques that violate coding standards or don't fully leverage known best practices.

Key Solution Requirements

The common thread among these 4 issues is a fundamental inability to access relevant information quickly and accurately. A variable solution to these key maintenance related problems must:

- Make it fast and easy for developers to access, explore and deeply understand highly interdependent code that is managed across many different systems and projects.
- Rapidly pinpoint where similar code is defined and used.
- Accurately determine defect injection points.
- Precisely predict defect propagation across multiple versions and projects.

- Quickly determine which projects will be impacted by code component, service (SOA) or third party code modification.
- Continuously discover violations of coding "best practices" before code changes are propagated.
- Quickly assess organization-wide exposure to known security and performance vulnerabilities.
- Identify candidates for re-factoring or SOA implementation.
- Detect and monitor unmerged code changes in components or shared code.
- Monitor code submits for known coding defects and compliance with key coding practices.
- Help developers more quickly and deeply understand the code they are working on, but which they did not author.

These requirements address remediation of the top maintenance issues as summarized in Table 2:

	Problem 1: **Learning Curve**	Problem 2: **Poor Planning**	Problem 3: **Duplicated Coding**	Problem 4: **Coding Compliance**
Explore code	✓	✓		
Locate similar code	✓		✓	✓
Identify defect injection points		✓		
Assess defect propagation		✓		
Company-wide Impact analysis	✓	✓	✓	
Coding practice monitoring	✓			✓
Global vulnerability assessment		✓	✓	✓
Refactoring, SOA opportunities		✓	✓	
Monitor code divergence		✓	✓	✓
Code related info access	✓			

Table 2. Maintenance Solution Requirements

Code Search Solutions to Maintenance

As mentioned earlier, the root problem behind the code maintenance issues is poor information access. Information access problems are a classic area for solutions that are driven by search technology.

Several issues make development (code) search complex and unique - these issues make traditional enterprise or "horizontal" search solutions ineffective.

Development information is stored in proprietary SCM and database systems that are related by a variety of proprietary metadata linking architectures. Being able to *continuously* and *reliably* access the code and then generate a searchable library is essential. Maintaining the project level code structure and preserving the information references to the code is also very important.

Another issue is the ability to support code related searches. Code searches and traditional keyword searches are fundamentally different. Being able to use syntactic and code structural characteristics of the code is an important way to pinpoint the right code. Similarly, the ability to detect highly similar code patterns is crucial.

One solution that addresses these capabilities is Krugle Enterprise. Two interesting and unique technologies are incorporated into this solution:

1. Comprehensive Code Inventory (CCI). Using its own patented technology, Krugle Enterprise automatically catalogs and indexes all of the relevant code contained within an organization's firewall. This creates a single, comprehensive up-to-date inventory of source code and related information for an organization. Information and links from resources related to the code are also indexed and fully searchable.

Krugle supports a wide range of SCM systems (including Clearcase, Perforce, CVS, SVN, StarTeam, etc.). Krugle makes complete, current and reliable code inventorying possible for the first time.

2. Krugle Enterprise delivers Code Optimized Code Search and Analysis – technology that helps pinpoint useful examples, reusable code and all instances of specified code patterns. Krugle's underlying index uses semantic information extracted from the structure of each code file. It also offers a query capability that helps match similar multi-line code patterns in addition to simple keywords. This enables accurate identification and location of code fragments and patterns.

Summary

Significant information access problems are responsible for cost and quality problems in software maintenance. These problems consume a significant portion of the budget in most software development organizations.

These problems can be addressed through search technologies that are optimized for source code and related development information. A viable technology must operate robustly across a variety of SCM systems and development environments, without creating operational risk or requiring changes to

existing software methodologies. A practical solution must also employ code optimized query capabilities that leverage non-code information as well as the source code.

References

[1]Annual software maintenance cost in USA has been estimated to be more than **$70 billion** (Sutherland, 1995; Edelstein, 1993). SOURCE: Software Maintenance Costs, Jussi Koskinen, Information Technology Research Institute, ELTIS-project. University of Jyväskylä, P.O. Box 35, 40014-Jyväskylä, Finland. Available at: <URL: http://www.cs.jyu.fi/~koskinen/smcosts.htm> , since 12th Sept. 2003. Email: koskinen@cs.jyu.fi)

[2] An average Fortune 100 company maintains 35 million lines of code (Müller et al., 1994)…Older languages are not dead. E.g. 70% or more of the still active business applications are written in COBOL (Giga Information Group)… There are at least 200 billion lines of COBOL-code still existing in mainframe computers alone (Gartner Group)…Lientz, B.P. & Swanson, E. (1980). "Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations". Addison-Wesley: Reading, MA, 214 p.

[3] Studies of software maintainers have shown that approximately **50%** of their time is spent in the process of *understanding the code* that they are to maintain (Fjeldstad & Hamlen, 1983; Standish, 1984). SOURCE: Software Maintenance Costs, Jussi Koskinen,