

Managing Code Within Distributed Groups

Code Management Mobility - “Pay Me Now or Pay Me Later” strategies

By Greg Spehar (spehar@full-knowledge.com)

Executive Summary

Defining a process such that multiple groups can share and update the same code base has been a problem in the software field for some time now. Several tools have been used to try to solve this problem. The most prevalent is the use of the source control systems to ensure that multiple people can work on the same code base. This works well if the changes to the code only require the use of internal files, but more often than not there is the problem of multiple changes happening to the methods being used by other sections of code and also changes to global symbols used by multiple methods. This dynamic is difficult to measure and manage as changes in one area have unknown and unintentional impacts in other parts of the software under change.

“Cooperation is required when source code changes to a software system involve the efforts of multiple programmers. In a medium-to-large software project (involving at least 100,000 lines of source code), there are several reasons why maintenance activities are rarely carried out by a single programmer. The sheer volume of the effort may be prohibitive. There may not be any single programmer with knowledge about the modules and the system sufficient to make the changes correctly and rapidly. There may be administrative considerations that make it inappropriate for one programmer from one managerial unit to modify the modules assigned to another programmer reporting to a different manager.

Instead, a group of programmers normally cooperates on a change that involves several modules, particularly a change that cuts across subsystem boundaries. When changes are being made in several modules, it is necessary for the changes to be consistent with each other. The changes must be *syntactically* consistent [15], meaning that the interfaces between the modules must be correct and that the modules can compile and link successfully.”

Automated Support for Software Maintenance and Evolution

<http://www.ece.utexas.edu/~perry/work/papers/icsm87.pdf>

G.E. Kaizer, PE Perry, ICSM 1997

November 1986

This change problem is usually managed by large systems with the use of interfaces such that only access to a section of code would happen at the interface level. The enforcement of this effort is one that most development groups have the most difficult time maintaining. This use of interfaces can protect sections of code from other sections of code if the provider of that interface holds that interface constant and also only uses interfaces that have been supplied by other sections of code.

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



“Over the past decade the software industry has experienced enormous increases in the quantity of software production. Unfortunately, we have not seen a similar increase in the quality of software or the productivity of programmers. As a result, society finds itself increasingly dependent on an ever-growing body of expensive software that is becoming increasingly unreliable, and this situation has occurred despite many concentrated research efforts in various areas such as software engineering, programming languages, and logic. Software today simply is not fundamentally more reliable than it was a decade ago.”

Modularity Matters Most
K. Crary, R. Harper, P. Leez, and F. Pfenningx
Carnegie Mellon University
October 31, 2001

http://www.itrd.gov/iwg/sdp/vanderbilt/position_papers/karl_crary_modularity_matters_most.pdf

Most development teams call these independent pieces of code, components. They tend to be anywhere from 10KLOC (KLOC is kilo (thousand) lines of code) to many 100KLOC in size. The most interesting aspect of this environment is the desire to have multiple teams that are either co-located or are located around the world. This “distributed” programming means that it is even more critical that the identified interfaces be held constant.

“When you introduce multiple components with different release cycles, problems become increasingly more difficult to address. One does not even need to introduce Remote Procedure Calls (RPCs) to manifest the problem, the issue can occur on the same machine, within the scope of a single process, and even with software all from one vendor.

One manifestation of this problem is popularly known as DLL hell. Installing or uninstalling one program may suddenly cause a seemingly unrelated program to fail. Inevitably, the root cause is a change to an interface. The traditional solution generally involves some sort of change management.

Distribute this same software over the scope of the internet and the problem rapidly becomes intractable. One simply can't rely on the ability to simultaneously upgrade all servers and clients with a snap of the fingers. To deal with this, old versions of clients need to work with upgraded servers. Depending on the topology of the software involved, the reverse may need to be true too.”


Intertwingly
Coping with Change

By Sam Ruby, March 15, 2002.

<http://www.intertwingly.net/stories/2002/03/15/copingWithChange.html>

If we have a system that has effective components that are independent of the other components we are able to move that code base from one group to another. This ability is due to the fact that the “new” team would only know what is required within that component. Knowledge externally is not required, such that if the component is effectively documented and partitioned the “new” team will be able to make changes to that code in a timely manner.

“Successfully bringing a product to market requires development teams to work in unison. Unfortunately, in many companies, individual development teams working on specific projects use their own sets of methodologies and tools—a decision that can have a huge effect on a company's ability to complete projects in a timely and efficient manner.



This problem is magnified for developers working at multiple sites because they are often unable to communicate and exchange software components or assets easily, which can increase development costs substantially. In addition, developers moving from one team to another must learn a new way of working, often reducing employee productivity and increasing the time required to complete a project.”

IBM Rational SCM solutions for distributed development
August 2004
Karen Wade
http://www-128.ibm.com/developerworks/rational/library/content/04September/3160/3160_scm-gdd.pdf

What this capability provides is the ability to move code bases from one group to the next within the same facility, within the same group but different locations and finally to move the code to an external company or entity that might be hired to perform maintenance. So the term “Code Management Mobility” can be coined.

“The abstraction of various components of software design into layers enables much more rapid development of new applications (standing on the shoulders of giants) by rapidly interconnecting previously tested software objects in new combinations. It also enables one to move code between various platforms with much greater liquidity. Because the underlying layers can be modified or replaced, so long as the “hooks” or interfaces to the upper layers remains the same, the same functionality is more easily achieved on a variety of disparate platforms.”

OPENING THE CODE: SOFTWARE EXCELLENCE AS A FUNCTION
OF ITS DEVELOPMENT ENVIRONMENT

A Thesis submitted to the
Faculty of the Graduate School of Arts and Sciences of Georgetown University
By: David S. McLaughlin, B.S.
<http://opensource.mit.edu/papers/mclaughlin.pdf>

This is an important topic since many companies are having to deal with ever growing code bases and need the ability to partition and hand out sections of code that the company may determine would be better served by another team or group.

This isn't a simple process, it includes the need to understand if the code is sufficiently partitioned to allow this to happen and if it is not sufficiently partitioned what needs to happen to “break” the target code base away from the crowd to make it “mobile”.

This paper will discuss in detail the required activities and tools necessary to effectively “brake” code from the crowd and maintain the code that now has “Management Mobility”.

The following will be answered:

1. What does it mean to “pay me now or pay me later”?
2. What is the financial impact of “pay me later”?
3. How does the process change to have Code Management Mobility?



4. What Static Analysis Tools are needed to perform Code Management Mobility?

Pay Me Now

The most challenging aspect of managing large code bases is the ability and more importantly the necessity of managing the interfaces between the large components of the system. If the system is sufficiently grand the importance of modularity even is more critical. As the system grows the importance increases to allocate the proper functionality into the right components and allow the right access to that functionality such that changes are applied in the proper location and utilize the system interfaces as they were designed.

Abstract

We contend that modularity is the key to improving software quality. We advocate a view of modularity that emphasizes not the mere assembling of software systems from component parts, but rather the specifications of interfaces between components, verification that components meet their specifications, and the assembling only of components with compatible specifications.

Key to this methodology is the use of types to specify and automatically to verify adherence to interfaces. We claim that this methodology makes a higher degree of software correctness possible than has often been achieved heretofore, and moreover, that it may be achieved in a practical manner. To reach this goal will require the development of sophisticated new type system, will require new techniques for modularizing certain correctness properties, and will require a delicate balance between concise code and automated checking.”


Modularity Matters Most
Karl Crary, Robert Harper, Peter Leez, Frank Pfenningx
Carnegie Mellon University
November 1, 2001

<http://www-2.cs.cmu.edu/~fp/papers/mmm01.pdf>

This usually does not happen in most organizations. The early implementation may have had an architecture and intent but as time passes and the application is updated, that intent or architecture is lost in the daily grind of adding more functionality at a high rate of speed. With the advent of ever-faster tools the ability to control the expanding architecture becomes more problematic. This results in a highly complex convoluted application that can be near impossible to properly update.

“The lessons learned from the second year of data reinforce the first year of data, that low quality products are difficult to maintain and will have a high(er) defect ratio in repair than well-designed products. This is not an earth shattering revelation, but sometimes one must have these data to prove the “pay me now or pay me later” rule holds for software just as well as for oil filters and auto engines. There were several significant changes in the processes (or enforcement of processes) in the following years.

First, the company realized the existing maintenance processes were not capable of better performance and instituted a major change in the testing process. EPs that were distributed beyond the site specific correction were subjected to longer testing. This delayed the availability of the correction, but is in agreement with the recognized defect to failure ratio relationship published by Adams (1984). Second, the company reduced the frequency of EPs. By taking a more conservative approach to distributing corrections (again consistent with the Adams' work, especially in the second or third year of corrective maintenance), it avoided taking site-specific corrections to a second site where they were not needed. This effectively reduced the recidivism



ratio to zero. (Since the middle of 1997, eight or more months of the year see 0 percent recidivism ratio). The change was so significant that the company was not able to repeat the analysis in the third year. The number of defective fixes was so low that it was not possible (or necessary) to use the same analysis methods.

Second, the company strictly controlled the development processes on the next release. It introduced defect-depletion tracking across the full development life cycle. It developed quality goals for the next release, and predicted and measured conformance to these goals (Weller 2000). The stated objective of development was “to put the SMT out of business” by reducing the STAR rates to the point where the volume would not sustain a separate maintenance organization.”

Software Quality Professional, Volume 3 Issue 1
Applying Quantitative Methods to Software Maintenance
ED WELLER

http://www.asq.org/pub/sqp/past/vol3_issue1/sqp3i1weller.pdf

In the end, in a highly coupled and complex system, a change can only be made as guess work and the team crosses their fingers and hopes the testing group does not find a problem that may have been injected. This leads to more problems as one problem leads to another due to the original change. So another change is made to fix the previous change and the team again sits on the edge of their seats hoping the test group does not find another problem. In the end, the team may feel like they are will never be able to release the software making the team pay the price of the shortcuts taken early on in the development process.

But as the following study showed, the addition of processes and tools can greatly reduce the problems created by changing the code.

“In summary, some of the more interesting observations that we made during our analysis include:

- Nearly 95 percent of all files in the software project were maintained at one time or another. If the common header and constants files are excluded from the project scope, we can conclude that nearly 100 percent of files were modified at some point in time after the initial release of the software product.
- Nearly 40 percent of the changes that were made to fix defects introduced one or more other defects in the software.
- Nearly 10 percent of changes involved changing only a single line of code; nearly 50 percent of all changes involved changing fewer than 10 lines of code; nearly 95% of all changes were those that changed fewer than 50 lines of code.
- Less than 4 percent of one-line changes result in error.
- The probability that the insertion of a single line might introduce a defect is 2 percent; there is nearly a 5 percent chance that a one-line modification will cause a defect. There is nearly a 50 percent chance at least one defect being introduced if more than 500 lines of code are changed.
- Less than 2.5 percent of one-line insertions were for perfective changes, compared to nearly 10 percent of insertions towards perfective changes when all change sizes were considered.
- The maximum number of changes was made for adaptive purposes, and most changes were made by inserting new lines of code.
- There is no credible evidence that deletions of fewer than 10 lines of code resulted in defects.”

Towards Understanding the Rhetoric of Small Changes
Proposed paper - MSR 2005: International Workshop on Mining Software Repositories
Ranjith Purushothaman and Dewayne E. Perry
<http://msr.uwaterloo.ca/papers/Purushothaman.pdf>

Pay Me Later

So now that we have established the impact of “Pay Me Now” let’s take a deeper look at the concept of “Pay Me Later” by defining the cost assessment based on the use of Static Analysis tools. In terms of understanding the impact of Code Management Mobility within this context we must assume the objective is to have potentially two objectives:

- 1) Allow highly trained team to manage the most complex code
- 2) Allow for lesser cost teams maintain code


For this discussion we will calculate the value of the second objective. To break apart the code base to allow differing skilled development teams to manage the code, which will save money on personnel, costs. The example that can be used is the following:

The focus will be on calculating the development costs 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 assumptions will be 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 assumptions and calculations are the following:

- 1) Calculation of Code Management Mobility Assumptions
 - a. Manpower per Component 400KLOC/20KLOC – 5 person/20KLOC
 - b. Assume developer cost is \$100K/year on high end - \$50K/year on low end
- 2) Code Management Mobility (Assume 20KLOC per Developer)
 - a. Clear Separation of Components (No Coupling)
 - i. Manpower per Component 400KLOC/20KLOC = 20 People
 - ii. Total Manpower for Product is 5 * 20 people = 100 People
 - iii. Two Components Managed by High Tier Development
 1. Two (2) * 20 people at \$100K year = \$4M / Year
 - iv. Three Components Managed by Lower Tier Development
 1. Three (3) * 20 people at \$50K year = \$3M / Year
 - v. Total Management Costs
 1. \$4M / Year + \$3M / Year = \$7M / Year
 - b. Coupling of Component Requires Single Integrated Team (All Coupled)
 - i. Assume costs as mentioned above
 - ii. Five Components Managed by High Tier Development
 1. Five (5) * 20 people at \$100K year = \$10 M / Year
 - c. Total Benefit for having Code Management Mobility
 - i. \$10 M - \$7 M = \$3 M
 - ii. Or a 30% improvement in manpower maintenance costs

So it is clear that a limited quality effort and a poor architecture during maintenance require one large group to continue the development. Where the “pay me later” will become the cost of maintaining the expensive group that



excludes the option of performing code management mobility. Additionally, at some point the complexity of the application will reach a point that even the experienced team will not be able to update the program effectively. This would force the management to rewrite the application. And at that point we can say, “Pay Me Later” has occurred.

Code Management Mobility Product Tools

There is a set of tools provided by Static Analysis Vendors that can provide “Code Management Mobility” analysis. These tools are based on advanced static analysis tools, which will make Code Management Mobility a real possibility for software development teams. This set of products require the following framework to perform their jobs properly:

- Static Code Analyzer and Reporting of Interface Violations – Static Analysis
- Visual Representation of Interfaces for Entire System – Architecture
- Desktop Static Code Analyzer to Prevent Interface Violations – Desktop Analysis
- Tracking Interface Defects through Time – Metrics Management

Each of these products are intended to supply the proper controls to allow the development team to identify interface issues at a high hit rate and triage/mine those messages quickly at the desktop or at a build. The following is a more detailed description of the tools.

Static Analysis

Build time tool that runs at every build or at an appointed build to provide a snapshot of the system interfaces at moment in time. The use of this build snapshot allows for the integration and execution of rules that will catch all interface/architectural defects, coding defects and code metrics of the standards that were agreed upon prior to the development effort. Static Analysis includes the most advanced static analyzer with the use of heuristics, interprocedural analysis and extensibility.

Architecture Tools

Architecture tool that allows architects to see the actual relationships between files, thus allowing architects to discover the real interface between two components. It also allows the architects to display the issues found by the Static Analysis tool in a graphical user interface. The architects can further use the tool to define an architecture that is enforced by each developer using Desktop Analysis and at every build using Static Analysis.

Desktop Analysis

The Desktop Analysis tool allows a developer level enforcement to ensure the developer does not violate the interface and defect controls for hard to find interface/defect bugs and security defects. The tools can also track and warn the developer when they have passed metric thresholds that will put their code at risk down the road. The enforcement of Architecture must happen at the developer’s desktop to gain the most immediate value. With the use of Architecture Tools the architectural intent can be translated to the developer such that they will not break the system level architecture interface laws that have been established by the team architects.

Metrics Management

Metrics Management is the tool that provides detailed metric data build over build. This tool will give the developer and managers the visibility and ability to gather information on many details including interface violations. Thus providing that information to the development managers to back up their development decisions when the time crunches occur. The executive team, having access to the same data, can also provide incentives that have clear metric objectives and goals allowing effective risk management.

Code Management Mobility Process Changes

The use of the tools needs to be integrated into existing processes or their needs to be additional processes defined to ensure the tools are used properly. Those process changes enacted by development roles will provide the mechanism for success. There are many ways to use these products to dig out of the immobility of coupled components. The following are just suggested implementations by product.

Static Analysis

Managers – Build reports detail the progress of the development effort, which can be reviewed
Architects – Monitor the build reports to identify architecture/interface violations at a system level
Quality Assurance – Monitor and review interface defects to make decisions on quality standards
TeamLeads/Developers – Review defects and schedule for updating code based on the reports

Architecture Tools

Managers – Architect Reports are generated to monitor the progression of the architecture/interfaces
Architects – Discover, Understand, Manage and deploy architecture/interface rules and decisions
Quality Assurance – Monitor and review architecture/interface to verify quality standards
TeamLeads/Developers – Monitor using Architecture Tools intent and interface decisions

Desktop Analysis

Managers – Monitors the development team to ensure the tools are performing
Architects – Performs periodic checks on the code using Desktop Analysis to ensure tools are performing
Quality Assurance – Monitors the usage of the Desktop Analysis tool by monitoring the Static Analysis Reports
TeamLeads/Developers – Enforcement tools used when making any changes to the code/interface

Metrics Management

Managers – Monitor churn rates and make development decisions based on trending and risk data
Architects – Monitor interface violation levels, complexity and risk points for indications of refactoring
Quality Assurance – Monitor and review interface levels, churn rates, complexity and risk hot spots
TeamLeads/Developers – Monitor the interface levels, metrics and trending to ensure compliance

Where these tools then are used by either the central system management site or distributed development site, the management at those sites will have the ability to guide the quality efforts that have been established. The central system management and core product/architect team can enforce and ensure the interfaces, processes and procedures are properly established, coupled with the design decisions and coding standards, the Static Analysis technology can verify and validate that the rules and procedures are used appropriately.

Now that the products and processes are better understood it is clear that an understanding of the value can be beneficial when making decisions for adopting a “Code Management Mobility” effort.