Adoption-Centric Software Maintenance Process Improvement via Information Integration

Shihong Huang

Computer Science & Engineering
Florida Atlantic University

shihong@cse.fau.edu

Scott Tilley

Department of Computer Sciences
Florida Institute of Technology

stilley@cs.fit.edu

Mike VanHilst

Computer Science & Engineering
Florida Atlantic University

mike@cse.fau.edu

Damiano Distante

Research Center on Software Technology
University of Sannio

distante@unisannio.it

Abstract

Software process improvement is an iterative activity, normally involving measurement, analysis, and change. For most organizations, the existing software process has substantial momentum and is seemingly immovable. Any change to existing process activities causes turbulence in the organization, which can be a significant barrier to adoption of the quality improvement initiative. This paper presents a quiescent, non-invasive, and adoption-centric approach to process improvement for software maintenance. The approach realizes the goal of improving the efficiency of existing processes by minimizing changes to existing workflows and focusing on integrating enhancements at the micro-level of the system. By leveraging information buried in existing data, making it explicit, and integrating the results with known facts, more informed decision-making is made possible. The approach is illustrated with a model problem concerning redocumentation of an embedded control system in the context of performing higher-quality software maintenance.

Keywords: adoption-centric, process improvement, software maintenance, information integration, process integration, program redocumentation

1. Introduction

The continuous maintenance and evolution of large-scale software systems is a constant challenge. The literature is replete with examples of the difficulty in dealing with issues of complexity management, program understanding, and impact analysis. One of the key drivers for system change is increased quality, which may be defined using somewhat ill-defined goals such as "easier to use," "less buggy," or "more secure."

Changes made to the system in response to shifting business requirements are often accompanied by changes to the processes used to manage the system. A software process is a set of activities that leads to the production of a software product [9]. Process improvement is based on the assumption that the quality of the engineering process is critical to product quality. This assumption has been proven to be true by Japanese manufactured goods. However, in the academic community, there are arguments whether the results from manufacturing engineering can be transferred directly to software engineering [14]. Nevertheless, process improvement is still viewed as an effective way to improve the quality of software engineering activities such as maintenance.

Although there are software process improvement techniques that have shown promise in limited settings (e.g., as an academic proposal or in small-scale controlled experiments), in reality, any approach to software process improvement should only be considered successful if it is adopted in a real-world context (and hopefully by the industry-at-large).

Proceedings of the 13th IEEE International Workshop on Software Technology and Engineering Practice (STEP'05)
0-7695-2639-X/05 $20.00 © 2005 IEEE

Unfortunately, making changes to an organization's processes is not a trivial task. For adoption to be successful it should be voluntary and facile; forcing the organization to change the fundamental manner in which they operate is rarely successful.

Many aspects of the existing software process activities, such as the current business environment, the familiarity of tools that software developers use, and the workflow the developers are accustomed to, may have existed for a long time. For better or worse, the workflow processes may have become deeply embedded in the organization's daily practice. Such processes exhibit considerable momentum, making change very difficult to effectuate.

To realize the goal of improving the quality of software maintenance, dramatic changes to existing processes is not a viable solution; it can only generate turbulence within the organization. A better way to achieve the same goal is to make the process improvement change transparent to the stakeholders, so that the adoption is a pleasant experience. One way to accomplish this is by integrating new procedures into existing process at the micro-level. The net effect is to positively change the results of the production process at the macro level, but with considerably less churn.

This paper presents a quiescent, non-invasive, and adoption-centric approach to process improvement for software maintenance. The approach realizes the goal of improving the efficiency of an existing process by minimizing changes to existing workflows and focusing on integrating enhancements at the micro-level of the system. By leveraging information buried in existing data, making it explicit, and integrating the results with known facts, more informed decision-making is made possible.

The next section discusses some of the fundamental issues related to software process improvement, outlining lessons learned as to what has worked and what has not worked over the years. Section 3 outlines an adoption-centric approach to process improvement, using a model problem concerning redocumentation of an embedded control system in the context of performing higher-quality software maintenance as a basis. Section 4 provides an illustrative example of the approach in use, highlighting key steps of the approach used in a real-world case study involving an industrial partner. Finally, Section 5 summarizes the paper and outlines possible avenues for future research.

2. Process Improvement

The accepted practice of software process improvement is to apply a standard framework, which is essentially a checklist of recognized software engineering best practices. A process "improves" as it comes into closer alignment with the elements and practices defined in the checklist. Some frameworks are applicable to the entire software lifecycle, while others are more focused on specific activities, such as maintenance and evolution. Some of the best-known process improvement frameworks are the Software Engineering Institute's Software CMM [11], ISO 15504 (SPICE) [10] and Trillium [1]. There are also many other process improvement frameworks, created by international standards bodies, the military, industry groups, and academic think tanks [13].

The common complaint about process frameworks is that they add activities rather than reducing or streamlining the process. Their tendency to emphasize documentation and inspections adds overhead to existing processes. When the prescribed activities run against the existing culture, an organization sometimes create a separate team to perform the additional work. This runs counter to the goal of adoption of the process improvement initiative, since the implicit lack of buy-in by the key personnel often proves to be an insurmountable obstacle.

Under the above definition of improvement, the more common notion of improving productivity is treated as a sub-area, called "optimization." There are fewer accepted practices specific to optimization. The best known, Primavera [12], the Personal Software Processes (PSP) [7], and Team Software Processes (TSP) [8], require frequent documentation of productivity metrics (e.g., time and results). These practices also tend to run against existing developer culture. Detractors of such optimization techniques sometimes compare them to time management, or even Taylorism [15].

Software process improvement is an iterative activity, normally involving measurement, analysis, and change. However, our experience with large organizations has been that process improvement is too often dictated from above, based only on superficial measurements and shallow analyses. Under pressure from customers and peers, high-level managers will dictate that the organization achieves certification under a particular standards framework. Under pressure to improve productivity, middle-level managers decide to adopt new tools and methodologies based on the vendors' claims of productivity improvement. At the

"bottom" level, the maintainers continue to work as they always have, ignoring any exhortations to adopt a new but unproven and heavyweight tool or technique.

At the process change stage, where the actual improvements take place, normal practice is to impose new procedures or introduce new methods and tools, and then attempt to integrate these changes with other existing activities. Such change is normally intrusive, and requires significant time and effort, including time off for training, to be properly introduced. There is a risk of the disruption creating a period of declining morale and product-quality degradation in the short term. In the long term, the improvement may not be adopted.

In contrast, a more effective approach should be compatible with other process activities and with organizational procedures and standards already in use at the organization – even if such practices can be shown to be suboptimal. Since the ultimate goal is to provide practical quality improvement within this constrained context, and to provide a better adoption approach of process improvement, reuse and transforming existing information seems to be a logical choice [16].

3. An Adoption-Centric Approach

The previous section identified several shortcomings in normal approaches to software process improvement. These shortcomings suggest possible changes to the existing methods by using an adoption-centric approach, so that the changed and improved process could be easily adopted by organizations with minimum ripple effect on the daily practice of the process. This section outlines such an approach, using our experience in developing a methodology for program redocumentation of embedded control systems as a representative model problem.

Modern real-time control systems are incredibly intricate hybrids of software components and hardware mechanisms. Gaining a sufficient understanding of such a system in order to perform reliable maintenance with predictable results can be quite difficult. One technique that has enjoyed considerable attention in this regard is reverse engineering [3], which is a process of gathering data through automated processing of the system (e.g., parsing source code) and generating abstract representations of the system to support program understanding. Reverse engineering can be used to retroactively provide accurate documentation for existing software systems. This documentation can then be used to aid the engineer in performing maintenance and evolution tasks.

The reverse engineering tools employed in support of maintenance of embedded control systems have unique requirements imposed on them due to the intricacies of the application domain. The unfortunate result is that many reverse engineering tools and techniques fail to be adopted by anyone other than their developers. This is a lamentable situation, since objective evidence as to the technique's efficacy is ultimately provided by the end user – preferably in a real-world setting [6].

To address this problem, we developed an integrated approach to program redocumentation that has adoption as one of its central themes [5]. The approach is prescriptive in nature: it provides general guidelines for developing a redocumentation process that has the ability to meet the user's requirements, and suggests manners in which selected tools can be integrated to realize the approach. The approach relies in part on reusing information that is hidden in the subject system – information that is uncovered and made explicit using reverse engineering.

The approach is structured according to the following five high-level steps:

1. Collect documentation requirements
2. Identify data sources and synthesis needs
3. Select tools according to an existing reverse engineering environment framework
4. Create a common schema
5. Develop a control integration strategy

Rather like the CMM's "key practices," each step of the approach describes what should be done, but it should not be interpreted as mandating how the process should be implemented. There is enough flexibility in the approach that each step can be tailored as needed.

Due to space requirements, this paper focuses on just one of the five steps of the approach most directly related to information integration: Step 2 - identifying available data sources that can readily provide the needed information. For program redocumentation, the likeliest candidate is source code. However, secondary and ancillary sources are also candidates. Examples of such sources include existing documentation, requirements specifications, design models, test cases, and maintenance histories.

Depending on the nature of the source, special-purpose data gathering engines will likely be required to extract what is needed. The approach does not specify the functional nature of the data-gathering engine, nor does it dictate details such as the internal representation used. The only requirement is the ability to extract the necessary data from the source and possibly blend the result in combination with other similarly gathered data.

If there are no appropriate data sources, or if the documentation requirements cannot easily be satisfied by single input streams, then the result must be synthesized. The synthesis is made possible by the common schema and loosely integrated tools described below. The synthesis may be as simple as combining the results from two queries against the database. Or, the synthesis may be a complex iterative activity that both uses and expands upon known information to derive the necessary insights into the application.

4. An Illustrative Example

The previous section outlined an adoption-centric approach to the process for a specific software maintenance activity: program redocumentation. The success of this approach rests on the underlying assumption of the importance of minimal invasiveness when introducing a new process into an organization. In other words, how difficult it is to integrate the instantiation of the approach into existing processes. This section again focuses on Step 2 of the approach: the identification of data sources and synthesis needs.

This illustrative example continues the use of an embedded control system as a representative problem in a complex application domain. With this particular system, the documentation requirements collection step identified three types of information that would be of particular use for the engineers: control flow and data flow documentation, functional structure documentation, and software architecture documentation. The first two types of documentation are accessible using a variety of data analysis techniques, while the third type of documentation required more elaborate processing and data synthesis.

The documentation requirements of control flow and data flow, functional structure, and software architecture consume various forms of data as input to the redocumentation procedure. This data can only come from two places: data that is readily available for gathering, and data that must be synthesized for use. Data analysis is relatively straightforward, although the actual task of processing the raw source may be non-trivial. If there are no appropriate data sources, or if the documentation requirements cannot easily be satisfied by single input streams, then the result must be synthesized. The synthesis is made possible by the common schema and a loosely integrated tool control strategy outlined above. The synthesis may be as simple as combing the results from two queries against the database. Or the synthesis may be a complex iterative activity that both uses and expands upon known information to derive the necessary insights into the application.

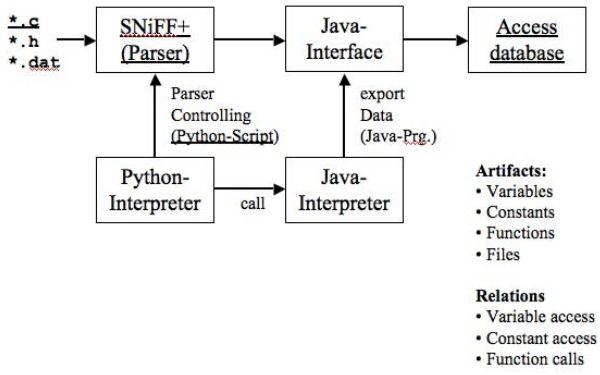

Figure 1: Control and Data Flow Analysis

4.1 Control Flow and Data Flow Documentation via Data Analysis

In the model problem, control flow and data flow documentation capture information related to two types of relations: function invocations and the usage of global variables by functions. Function invocations are essentially the def/use relationships that are extracted from the source code. The variable usage is classified as "read," "write," or "readwrite." The exact nature of the variable usage is determined by source code analysis. Depending on the capability of the tool used to process the system's source code, this information may be readily available, or it may require further analysis to synthesize the result.

For the model problem, the main source of data was the control system's C source code. This is illustrated in Figure 1. The following types of artifacts were gathered from the source code: variables, constants, functions, files, label access, and function calls. Bit variables were extracted from the source code, augmented with manually processed information and database analysis routines (as described below). The code is truly "real world" in the sense that it makes use of programming language constructs that pose analysis difficulties (e.g., macros).

Existing documents were also used as input to the data gathering process. Writing documentation means collecting information, making investigations, combining this information with other sources, inferring new facts, understanding the whole, and generating

updated documentation. Therefore, creating new documentation can be interpreted as the reengineering of existing documentation. Similarly, redocumentation can be viewed as the same process as regular documentation, only the data sources are different. This makes adoption of the redocumentation process into existing processes even more important: the new information must be produced in an unobtrusive manner.

The third form of document data gathering is incorporating user input. Senior engineers often carry implicit knowledge concerning the system, such as why particular design decisions were made. By making this knowledge explicit and storing it in the same repository as the other two sources of data (source code and existing documents), a repository is created that captures the essential documentation building blocks that are used later in the redocumentation phase.

There are also secondary data sources that contain important information regarding the system's requirements, design, and construction written in a variety of other non-programming language formats. For example, information regarding real-time objects such as tasks, processes, and interrupt service routines was contained in a semi-structured text file. A coarse description of the system's architecture was written by hand in an Excel spreadsheet by one of the system's original developers.

Even with such a rich variety of data sources available, not all information was readily present to support all types of documentation requirements. For example, for the functional structure documentation, there was a need to connect logical information from one data source to code information from another. Perhaps the most challenging requirement was bit-level data flow analysis, which required extensive synthesis of information already in the central repository to create the necessary documentation.

4.2 Functional Structure Documentation via Data Analysis

In the context of the model problem, "functional structure" refers to a representation of the functional components of the system—in particular, to its logical structure according to engineer-defined "functionality." From a traditional software engineering perspective, the term "functionality" may be counterintuitive. It does not directly refer to functional requirements. Rather, it refers to a logical clustering of sub-functionalities (a recursive definition) and files that constitute a logical component.

The functional structure is hierarchical in nature and best modeled as a layered graph, where a node in the graph can be either a sub-graph representing sub-functionality or a terminal node representing a non-decomposable entity. The advantage of using a layered graph with "super nodes" that represent sub-graphs is that this model seems inherently in tune with how developers think of the system's functional structure. This implies that an important criterion for tool selection is support for hierarchical displays of information, preferably in an interactive manner. Without this ability, any graphical representation of nested structure would be severely compromised.

To provide functional structure documentation, there must be a way of extracting information from the system that indicates the structure of the "functionalities" (the term used in the project for the representation of the system's functional components). For most programs, this information is not available in the source code. Instead, secondary sources of information must be used to provide this data. This requirement implies that traditional forms of data gathering based on static source code analysis will not be sufficient. Once the functional structure data has been gathered, there must be a way to merge this data into the central repository to facilitate later analysis. This can be accomplished (in part) through data synthesis.

4.3 Software Architecture Documentation via Data Synthesis

Most architecture recovery tools rely solely on static source code analysis. But the rich architectural information is not codified in source; it is usually kept as mental models in the engineers' heads, or documented in an informal manner. This makes data gathering and subsequent analysis of this information quite challenging, imposing special requirements on the toolset used for the process.

As with functional structure analysis, there must be a way of extracting data from the system that carries some value in terms of architectural structure. As mentioned before, for most programs, this information is not available in the source code. Instead, secondary data sources must be used to provide this information. Moreover, there must be a way to merge this data into the central repository to facilitate later analysis.

For the project, architectural information was kept in semi-structured documents such as Excel spreadsheets, which implies that analysis of this information must be

blended with other sources to create a complete picture of the system.

5. Summary

This paper presented a quiescent, non-invasive, and adoption-centric approach to process improvement for software maintenance. The underlying thesis is that improvements to the software process will produce measurable improvements to the resultant software product. Making process improvements of this scope have proven notoriously difficult to properly execute.

To address this historical deficiency, the approach realizes the goal of improving the efficiency of an existing process by minimizing changes to existing workflows and focusing on integrating enhancements at the micro-level of the system. By leveraging information buried in existing data, making it explicit, and integrating the results with known facts, more informed decision-making is made possible.

The approach was illustrated with a model problem concerning redocumentation of an embedded control system in the context of performing higher-quality software maintenance. Software engineers have long relied upon documentation as a valuable source of information when it comes to making informed decisions regarding potential changes to the application. The approach described has the ability to blend data from multiple sources to create a more complete documentation package.

5.1 Results

Although this paper focused on just one step of the approach, the entire procedure has been implemented and deployed at an industrial site, and has been in continuous use for over three years. Because the resultant toolset and the software maintenance process that it supported were relatively easy to integrate into existing workflows, the system was adopted with little negative impact on daily activities. Indeed, the positive impact of the program redocumentation activity was significant: information previously unobtainable was made available to support software maintenance activities in an automated manner.

The program redocumentation procedure can be initiated on-demand by a system engineer, to produce timely information as part of normal maintenance activities. Since the system was also incorporated into the nightly build process, extra information is automatically produced and made available for use on as as-needed basis the following workday [4].

Although this was just one case study, the results were encouraging and seem to support the central hypothesis that process improvement initiatives, whether they are focused on general activities or specific ones such as maintenance and evolution, must be adoption-centric from the very beginning; otherwise, they will have little measurable (positive) effect on the target organization.

5.2 Future Work

One of the most promising areas of future work concerns incorporating new sources of semi-structured data into the redocumentation process. These sources are rich in informal history and shared experience by past project members. However, such sources can also prove to be difficult to analyze in an automated manner. Fortunately, for program redocumentation purposes the analysis need not be exact; it just needs to provide new information not otherwise available to the decision maker.

There is also a clear need for continued use of evidence-based methods [2] to measure the efficacy of the approach. As with many other developments in software engineering, the user is the ultimate judge as to the value of a new tool or technique. When the techniques concern changes to incumbent processes, a key to adoption appears to be minimally-invasive alterations in the status quo with maximally-observable benefits to the encompassing production cycle.

References

[1] Bell Canada. *TRILLIUM - Model for Telecom Product Development and Support Process Capability*. 1994.

[2] Budgen, D.; Brereton, P.; Kitchenham, B.; and Linkham, S. *International Workshop on Evidence-Based Software Engineering* (EBSE 2005: May 17, 2005; St. Louis, MO). In *Proceedings of the 27th International Workshop on Software Engineering* (ICSE 2005: May 15-21, 2005; St. Louis, MO), pp. 687. New York, NY: ACM Press, 2005.

[3] Chikofsy, E.; and Cross, J. "Reverse Engineering and Design Recovery: A Taxonomy." *IEEE Software* 7(1):13-17, January 1990.

[4] Hartmann, J.; Huang, S.; and Tilley, S. "Documenting Software Systems with Views II: An Integrated Approach Based on XML." *Proceedings of the 19th Annual International Conference on Systems Documentation* (SIGDOC 2001: Santa Fe, NM; October 21-24, 2001), pp. 237-246. ACM Press: New York, NY, 2001.

[5] Huang, S. *An Integrated Approach to Program Redocumentation*. Ph.D. Dissertation, University of California, Riverside. 2004.

[6] Huang, S. and Tilley, S. "On the Challenges in Fostering Adoption via Empirical Studies." *Proceedings of the 4th International Workshop on Adoption-Centric Software Engineering* (ACSE2004: May 25, 2004; Edinburgh, UK).

[7] Humphrey, W. *Introduction to the Personal Software Process*. Addison-Wesley, 1996.

[8] Humphrey, W. *Introduction to the Team Software Process*. Addison-Wesley, 1999.

[9] Humphrey, W. *Managing the Software Process*. Addison-Wesley, 1989.

[10] ISO/IEC. "Information Technology - Software Process Assessment." ISO/IEC Technical Report TR-15504 1998.

[11] Paulk, M.; Weber, C.; and Curtis, B. (eds). *The Capability Maturity Model: Guidelines for Improving the Software Process*. Addison-Wesley, 1995.

[12] Pressman, R. *Software Engineering: A Practitioner's Approach* (6th Ed). McGraw-Hill, 2004.

[13] Sheard, S. "The Frameworks Quagmire." *Crosstalk*, September 1997. Online at www.stsc.hill.af.mil/crosstalk/1997/09/frameworks.asp.

[14] Sommerville, I. *Software Engineering* (7th Ed.). Addison-Wesley, 2004.

[15] Taylor F. *The Principles of Scientific Management*. 1911.

[16] VanHilst, M; Garg, P; Lo, C. "Repository Mining and Six Sigma for Process Improvement." *International Workshop on Mining Software Repositories* (MSR'05: St Louis, MO; August, 2005).