Quantifying the Value of New Technologies for Software Development

D. L. Atkins, A. Mockus, and H. P. Siy

Abstract: Introducing relevant software technologies may provide significant advantages to a software organization. Unfortunately, the value the technology may provide is almost never quantified. We describe a methodology for precise quantitative measurement of the value a software technology may add to the project in terms of the impact on quality and lead time. The methodology employs measures derived from version control and problem tracking repositories to determine the value of technology. We illustrate this approach in a detailed case study on the impact of using two particular technologies - a version-sensitive source code editor and a domain engineered application environment - in a telecommunications product. In both cases use of technology had a strong positive impact on the considered quality measures. The methodology relies on information commonly available in project version control and problem tracking systems and, therefore, can be widely and easily applied.

Keywords: statistical models, empirical studies, case studies, software change database, software metrics, software quality, development lead time, version-sensitive editing, domain engineering, large scale software development, technology transfer, technology evaluation.

Introduction

New technologies—languages, tools, methodologies—are constantly being introduced in the hopes of improving quality, decreasing lead time, or increasing productivity. While they have the potential to greatly improve the quality and maintainability of software, deploying and maintaining a new technology in a large organization can be an expensive proposition. We explore how to quantify the effects of assimilating software engineering technologies into ongoing large-scale software projects, presenting a simple methodology that correlates technology usage with field defects and lead time based on analysis of the change history of a software project.

Quantifying the impact of a technology on software development is particularly important in making a case for transferring new technology to the mainstream development process. Technology transfer involves significant effort spent in training developers and integrating the technology to the existing development process. It also carries the risk of decreasing developer productivity due to the inevitable learning curve. (Rogers, 1995) cites observability of impact as a key factor in successful technology transfer. Observability usually implies that the impact of the new technology can be measured in some way. Most of the time, the usefulness of a new technology is demonstrated through best subjective judgment. This may not be persuasive enough to convince managers and developers to try the new technology. By having a methodology for quantifying the value added by the new technology, early adopters can be assured that an objective evaluation can be performed after trying it out. Furthermore, having quantified results from other projects gives interested practitioners an opportunity to gauge whether the new technology has potential for a positive return on investment in their environment.

Previously we have reported on a methodology to estimate the savings in terms of effort to perform a software change provided by new technologies in (Atkins, et al., 2000, Atkins, et al., 2002). We now extend this methodology to also estimate the impact of the new technology on defect and lead time reduction—two qualities that are likely to prove more valuable (Boehm, 2003) to a software organization than developer effort savings. Furthermore, instead of estimating these two qualities for individual software changes, we estimate these for units that add business value. In our case, these are called features—customers buy products or upgrade to new software releases contingent on the on-time delivery of certain features that have passed their rigorous acceptance tests.

While still focusing on the analysis of changes to the software, our estimation methodology is modified accordingly to deal with features. First, we obtain a number of change measures, such as size, lead-time, and technology usage, from the change history of the source code. Then we add a new step where we aggregate changes into their associated features. Finally, we fit statistical models that relate defects and lead-time in the considered units to the predictor that indicates usage or non-usage of the technology. We also have an additional four years of data to verify the trends observed in previous reports.

As we will see, the methodology is largely automatic, inexpensive, non-intrusive, and applicable to most software projects using version control systems. Furthermore, it can be applied to an entire software project in its actual setting as we do here to measure the effects of a version-sensitive source code editor and of a domain engineered application environment. Despite fairly simple general features, there are a number of differences between the ways the methodology is applied to estimate the impact of various technologies. The goal of this paper is to highlight and summarize these differences to make the methodology easier to use in practice.

We start by briefly describing the software project under study, software changes, and data sources in Section 2. Section 3 describes the two technologies under consideration. Section 4 describes the step-by-step application of our methodology. Finally, we conclude with a relevant work section and a summary.

Background

The case study here revolves around a large telephone switching software system developed over more than two decades. Lucent Technologies' 5ESS® switch is used to connect local and long distance calls involving voice, data and video communications. The 5ESS source code is organized into subsystems with each subsystem further subdivided into a set of modules. Each module contains a number of source code files. The change history of the files is maintained using the Extended Change Management System (ECMS) (Midha, 1997), for initiating and tracking changes, and the Source Code Control System (SCCS) (Rochkind, 1975), for managing different versions of the files.

We present a simplified description of the data collected by SCCS and ECMS that are relevant to our study. ECMS, like most version control systems, operates over a set of source code files. An atomic change, or delta, to the program text consists of the lines that were deleted and those that were added in order to make the change. Deltas are usually computed by a file differencing algorithm (such as Unix diff), invoked by SCCS, which compares an older version of a file with the current version.

ECMS records the following attributes for each change: the file with which it is associated; the date and time the change was "checked in"; and the name and login of the developer who made it. Additionally, the SCCS database records each delta as a tuple including the actual source code that was changed (lines deleted and lines added), login of the developer, MR number (see below), and the date and time of change.

In order to make a change to a software system, a developer may have to modify many files. ECMS groups atomic changes to the source code recorded by SCCS (over potentially many files) into logical changes referred to as Modification Requests (MRs). There is one developer per MR. An MR has an English language abstract associated with it, provided by the developer, describing the purpose of the change. A timestamp of when the MR was opened is also recorded in ECMS.

We also obtained a complete list of identifiers of MRs that were done using the domain engineered application environment and/or using the version sensitive editor. Thus, for each MR, we were able to obtain the following information:

- who made the change (developer login)
- size of the change (number of lines added and deleted)
- number of deltas
- duration (dates of first and last deltas)
- indicator if the change was done to fix a problem in a released version of the software
- number of files touched
- whether the change was done using the technology under consideration.

Applications

In this section we describe two technologies we evaluate. The first one represents a source code editor that is designed to show a desired version of the source code. The second example describes a domain engineered application environment including a special language and a GUI based code generator.

VE: A Version-sensitive Editor

The Version Editor (VE) is used by 5ESS developers to simplify the view of source code as they make changes. The software project for these programmers requires the concurrent development and maintenance of many sequential versions as well as two main variants for domestic and international configurations of the product (Perry, et al., 2001). The 5ESS source code may be common to more than two dozen distinct releases of the code, which may be deployed products in maintenance mode, or new product versions under active development.

As described in (Atkins, et al., 2002), the software releases form a complex version hierarchy with the often conflicting project management goals of isolating deployed releases from current development changes yet maximizing commonality to promote the automatic flow of software fixes to future releases. The implication is that, at any given time, several releases of the software are in the field and are actively being supported. Several versions of the source code needed to be maintained. Since the industrial source code management technology of the early 1980's did not have good support for branching and merging, source code was kept common among many releases with release specific differences delineated by a special embedded #version directive. This directive is similar to a C prepro-

Before . . .

```
 routing = getRoute(routing);
#version (4A)
 dest = getDest(routing);
 if (dest.port == 0)
   return (ConnectLocal(routing));
#endversion (4A)
 Connect(routing);
```

After . . .

```
 routing = getRoute(routing);
#version (4A)
 dest = getDest(routing);
#version (!5A)
 if (dest.port == 0)
#endversion (!5A)
#version (5A)
 if (dest.port == 0) || dest.module == 0)
#endversion (5A)
   return (ConnectLocal(routing));
#endversion (4A)
 Connect(routing);
```

Figure 1 – Before and after a Release 5A change. Emboldened lines are the code added by the programmer.

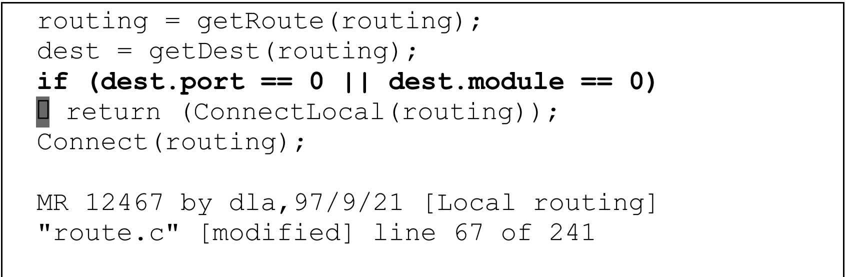
cessor #if where a symbol (corresponding to the release) is used for control and the symbol may be negated.

This system permits a single source file to be extracted to produce a different version for each software release. Software development environment tools verify the consistent use of these constructs according to a release hierarchy maintained by the system and perform the extraction of the source code for building each software release. For example, the first frame in Figure 1 shows a source file where three lines of code are specific to the 4A release. The system guarantees that these lines will not appear in earlier releases but will appear in later releases. Also, the lines will not appear in isolated releases (the domestic and international configurations are all isolated from each other).

A developer adding new code must target the change for a specific release and then bracket it by the appropriate #version constructs. When existing code is changed, it must be logically deleted with a #version construct using the negation of the target release. Figure 1 shows how these constructs are used to change the expression in an if-then statement for Release 5A. The original if-then statement was code inserted for Release 4A.

This simple example shows how even a one line code change requires the developer to add five lines to the file (four control lines and the changed code line). In addition to this extra overhead for a logical one line code change, the version control lines make the source file more difficult to read and understand. In the project being studied there are several dozen distinct releases and some core source files may contain #version directives for most of these releases. In worst case files, only 10% of the lines of the file are the extractable source code for a release, with 50% of the lines being #version/#endversion lines and the other 40% being source that extracts for other releases.

The version-sensitive editor VE (Coplien, et al., 1987, Pal and Thompson, 1989, Atkins, 1998) was made available to make this situation more manageable for the developer. This tool allows the developer to edit in a view that shows only the code that will be extracted for the release being changed and performs the automatic insertion of any necessary control lines.



```
routing = getRoute(routing);
dest = getDest(routing);
if (dest.port == 0 || dest.module == 0)
▮ return (ConnectLocal(routing));
Connect(routing);

MR 12467 by dla,97/9/21 [Local routing]
"route.c" [modified] line 67 of 241
```

Figure 2 – Release 5A view in VE with change in bold

The developer’s view is of normal editing in the extracted code; VE manages the changes to the #version constructs according to the described constraints. Figure 2 shows the view presented by VE for the file from Figure 1. The developer only has to use standard vi or emacs editing commands, and VE inserts the required #version directives (behind the scenes).

The use of VE by developers is entirely optional. The usage of VE may be detected, because VE leaves a signature on all of the #version/#endversion control lines that it generates. (See (Atkins, et al., 2002) for more details.) Thus we can distinguish when VE was used to make a change involving #version lines from when the change was made using an ordinary editor.

Figure 3 shows the history of VE usage in the considered project, which consists of approximately 1.2M MRs. The three lines show the fraction of MRs that were done with VE (V: MRs such that at least one delta of the MR contained #version lines with the VE signature), that involved #version line (F: MRs such that some delta of the MR contained a #version line), and fraction of #version MRs that involved VE (%: V/F). The usage of VE increased dramatically over time.

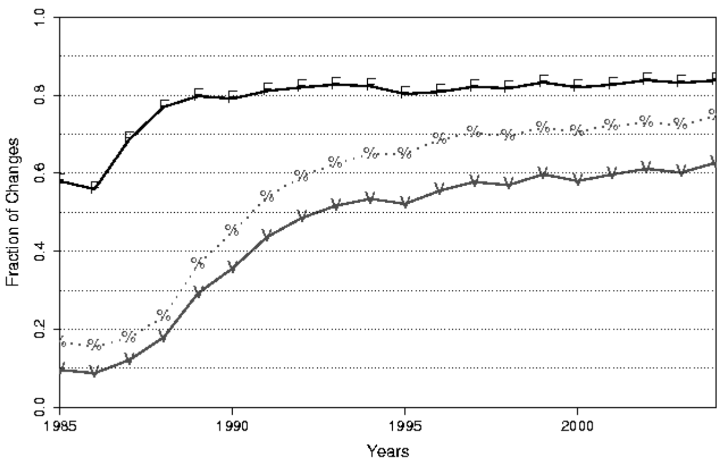


Figure 3: VE usage over time.

Figure 4 shows the history of VE usage in terms of the fraction of developers that use it. The three lines show the fraction of developers that used VE (V: developers such that at least one delta within a year contained #version lines with the VE signature), that made changes with #version line (F: developers such that some delta within a year contained a #version line), and ratio of the quantities above (%). The figure indicates that 60% of developers make changes involving #version lines and 70% of them use VE.

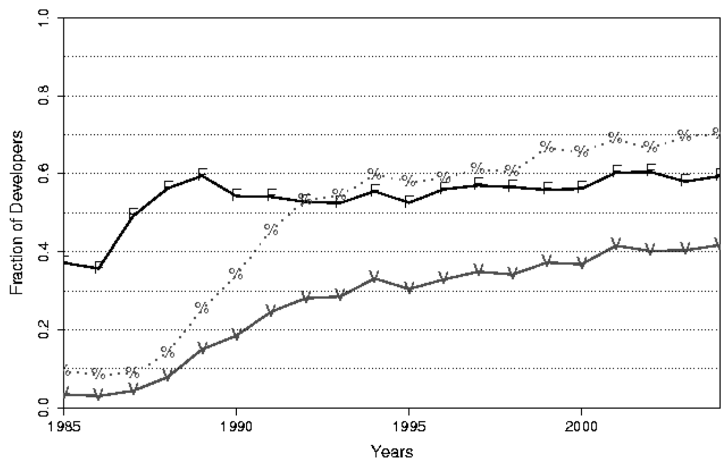


Figure 4: VE usage over time.

Domain Engineering

Traditional software engineering deals with the design and development of individual software products. In practice, an organization often develops a set of similar products, called a family or product line (Weiss and Lai, 1999). Traditional methods of design and development don't provide formalisms or methods for taking advantage of these similarities. As a result the developers practice some informal means of reusing designs, code and other artifacts, massaging the reused artifact to fit into new requirements. This can lead to software that is fragile and hard to maintain because the reused components were not meant for reuse.

Domain Engineering (DE) (Weiss and Lai, 1999, Coplien, et al., 1998, Cuka and Weiss, 1998) approaches this problem by defining and facilitating the development of software product lines rather than individual software products. This is accomplished by considering all of the products together as one set, analyzing their characteristics, and building an application engineering environment to support their production. In doing so, development of individual products (henceforth called Application Engineering) can be done rapidly at the cost of some significant up-front investment in analyzing the domain and creating the environment.

The process is summarized in Figure 5. In this figure, DE is further divided into domain analysis and domain implementation and integration. Domain analysis identifies the commonalities among members of the product line as well as the possible ways in which they may vary. Usually, several domain experts assist in this activity. Also, the application engineering environment is designed and built. This usually involves creation of a domain-specific language, a graphical user interface front end, and a source code generator back end. Domain implementation and integration deploys the DE-based process, making necessary adjustments to product construction tools (makefiles, version control systems, etc.) and to the overall development process.

Several teams have used the DE-based process to reengineer specific domain areas within the 5ESS software (Ardis and Green, 1998). We conducted a study to evaluate the impact of the AIM project, a DE effort to reengineer the software and the process for developing the multiplicity of screen interfaces to the 5ESS switch database.

Figure 5: Domain Engineering-based development is an iterative process of conducting Domain and Application Engineering.

The problem faced by the screen developers was that most clients who purchased the 5ESS switch required customization of their screen interfaces. In the old process, screens were customized by inserting #ifdef-like compiler directives into existing screen specification files. Over time, the specification files have become difficult to maintain and modify.

The AIM project used DE to identify commonalities and variabilities in different clients' interface requirements. These results provided input to the development of a GUI tool for assisting in the design of and keeping track of the customized screens. Information gathered through the GUI was saved in files whose format was specified by a domain-specific language. During the product build process, a code generator would then take these files and generate the screen specification files.

More details on the AIM study is published in an earlier paper (Siy and Mockus, 1999). In some sense, the problem here is not unlike the problem addressed by VE which facilitates the maintenance of multiple versions of code. However, the creators of AIM undertook a higher level, domain-specific solution in an attempt to achieve even higher productivity.

Impact Assessment Methodology

We outline here a general framework for analyzing the impact of a software technology. We have previously investigated effects on effort spent on individual changes (Atkins, et al., 2000, Atkins, at al., 2002). Because the technology may affect the definition or granularity of changes and also quality and lead time, here we focus on modeling the lead-time and quality impact on software features, the units that provide added value to software by providing additional functionality that may be compelling to the customer and provide revenue to the software provider. More specifically, features add value to the software because they generate revenue and enhance competitiveness of the product. We assume that on average, all features implement a similar amount of value. This is a reasonable assumption since we have a large number of features under both conditions and we do not have any reason to believe that the definition of a feature changed over the considered period. Consequently, even a substantial variation of functionality among features should not bias the results. A more precise measure of impact could be obtained by assigning weights corresponding to the actual or projected revenue corresponding to each feature. To approximate such revenue we used the size or complexity of the feature.

The analysis framework consists of the following steps:

1. Obtain measures of changes. Identify the changes made to the software entity of interest and whether or not the technology was used.
2. Group changes into software features or other relevant units that add value. The grouping also involves rolling up the measures of individual changes to the feature level.
3. Select a subset of these rolled-up measures to predict feature quality and lead time. The minimal subset typically includes the size of the change and an indicator as to whether the technology was used or not. Verify independence of predictors.
4. Fit and validate a set of candidate models. Models that explain more variation in the data and have fewer parameters are preferred. Our goal is to select simple models with predictive power rather than complicated models that account for all the variations of the response variable but are difficult to interpret. The fitted models are used to test the significance of the effect of technology.

The following sections explain each step in detail.

Change Measures and Technology Use

The basic characteristic measures of software changes include: identity of the person performing the change; the files, modules, and actual lines of code involved in the change; when the change was made; the size of the change measured both by the number of lines added or changed, the number of deltas, and the number of files touched; and the purpose of the change including whether the purpose of the change was to fix a field defect. Many change management systems record data from which such measures can be collected.

The information on files, modules, and lines changed is usually sufficient to determine if the software entity of interest was touched by the change. The determination of technology involvement in the change might be more complicated. We first discuss how to determine if the technology was used and then if it was not used.

In real life situations developers work on several projects over the course of a year and it is important to identify which changes they performed using the technology of interest. There may be several ways to identify these changes. In our VE example the tool left a trace in the SCCS files. In the AIM example the domain engineered features were implemented in a specific set of code modules (we refer to them as AIM paths).

Finally, to perform the comparison, we need to identify changes to a software entity that were done without the use of the technology. In the case of VE the information was available directly from SCCS except for a subset of changes that had no #version lines. Consequently we had two types of MRs: changes done using VE and changes done without VE. In the AIM example, the source code to the previously used screen specification files had a specific set of directory paths. We refer to those paths as pre-AIM paths. Based on AIM and pre-AIM sets of paths we classified all AIM MRs into two classes: MRs that touched at least one file in the AIM path and MRs that do not touch files in the AIM path, but touch at least one file in the pre-AIM path. In both cases there are two categories of changes that we label:

- TECH: MRs on the software entity that involve use of technology;
- no-TECH: MRs on the software entity not involving the use of technology;

We excluded features where technology could not be used (code not relevant to AIM functionality) or could not provide benefit (changes with no #version lines).

Aggregating change measures

Since our primary concern is to assess the technology impact on software value, we need to combine software changes into groups, each of which is providing comparable value. In the considered organization such groups were referred to as software features. Each feature was designed to provide functionality that could be sold. While the software code was common to all customers, only the licensed features were enabled.

Therefore, we wanted to measure technology impact on defect and lead-time reduction on each feature. Because larger and more complex features may take more time and have more defects, we may need to adjust for their size and complexity better to discern the effects of a technology.

To measure feature size and complexity we aggregate the MR measures to the feature level:

1. NMR - number of MRs
2. NDelta - number of delta
3. NLOC - number of lines added
4. NDEV - number of developers who participated
5. NFILES - number files modified
6. whether or not there were changes involving technology use
7. interval from first to last delta
8. if there were MRs fixing field problems

The last two measures were our response variables measuring lead-time and presence of field problems.

Variable Selection

Naturally, the size and complexity of a change may have a strong effect on the lead-time or probability of a fault. In the case of VE, such covariates were included because there is no reason to assume that the use or non-use of VE affects the number or complexity of the changes needed to implement a feature. Thus, we chose the number of developers, the number of MRs and the number of added lines as the covariates for predicting feature lead-time and quality. We used Spearman correlations due to the highly skewed nature of the observed data. Other measures we collected had correlations above .8 with the number of developers making interpretation of the regression results difficult. The correlation between these three measures and the indicator of VE usage ranged from .2 to .3.

In the AIM case, the programming language was different. Additionally, the changes involving technology were done using a special GUI environment instead of editing the source code in individual files. These reasons suggest that the number, size or complexity of changes to implement a feature would vary depending on whether or not AIM technology was used. Furthermore, due to previously reported dramatic effort savings, fewer developers may be needed to implement a feature. Therefore, inclusion of change size and complexity covariates may not be applicable when measuring the impact of AIM. After all, AIM was designed to simplify and streamline the changes. Thus, we did not include any covariates in the AIM models.

Models, Interpretation, and validation

In this step, we are ready to fit the models and interpret the results. Due to the highly skewed nature of the software change data it is important to transform all three predictor measures and the lead-time response variable via logarithms. The presence of the fault is such a rare event that we modeled it as a boolean variable (zero or not). For the lead time we use multiple linear regression and for the faults we use logistic regression suitable for the binary response variable.

It is essential to validate software repository data. See, for example, (Atkins, et al., 2002, Herbsleb and Mockus, 2003, Mockus, et al., 2003, Mockus and Votta, 2000, Mockus and Weiss, 2003) for more details. The key is to understand and validate how the derived attributes of changes relate to the actual software process and exclude computer-generated or data collection artifacts. It is important to have several operationalizations of a measure and check for consistency among them.

The statistical aspects involve using appropriate transformation of the variables, excluding strongly correlated predictors, and using appropriate statistical models and procedures.

Other aspects of validation include realization that some technologies may impact the change measures directly, in addition to affecting the outcome variables as happens to be the case with AIM. Finally, the external validation of measures and estimates is performed by presenting and discussing the results with the organization and individuals involved in the study.

Results

We present the technology impact on feature lead-time and quality. We start with the lead time, then investigate quality, and, finally, inspect the hypothesised AIM impact on the number of individuals that are needed to implement a change.

Feature lead-time

Our response variable is the natural logarithm[1] of calendar time between the first and last delta in a software feature. We exclude infrastructure features that are not "sellable" to customers but are an integral part of the system because they add a different type of value that may be impossible directly to express in terms of additional revenue.

This response variable represents development lead-time, which can be automatically collected from system repositories. We chose this part of the total interval because development lead-time is most likely to be affected by the technologies we are evaluating. To validate such measure of lead time, in previous invest-

[1] All logarithms in this chapter use the natural logarithm function.

igations of the same product we compared a sample of such automatically derived development lead-times with the total lead-times reported in project management records and found strong and consistent relationship where the total lead time was a constant multiple of the automatically derived development lead-time measure.

The predictor variables are the use of technology and the applicable covariates in case of VE application. The regression formulas are as follows:

$$E(\log time) = \alpha + \beta_1 \log NDev + \beta_2 \log NMR + \beta_3 \log NLOC + \theta_{VE} \quad (1)$$

$$E(\log time) = \alpha + \theta_{AIM} \quad (2)$$

In these formulas, we use θ_{TECH} (where TECH is AIM or VE) as a shorthand for I(TECH)θ_{TECH}, where I(TECH) is 1 if the feature involves the use of technology and 0 otherwise.

Table 1 presents the results of the regression using formula (1).

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	12.54	0.04	303.76	0.00
log(NDEV)	0.46	0.02	19.68	0.00
Log(NMR)	0.21	0.02	12.54	0.00
log(NLOC)	0.23	0.01	30.50	0.00
VE	-0.10	0.03	-3.64	0.00

Table 1: Feature lead-time regression, VE impact. 15953 features, $R^2 = .4$.

These estimates indicate that the lead time for a feature with median number of developers (3), median number of MRs (3), and median number of lines (725) is 11% longer when VE was not used. Not surprisingly, larger features with more developers, MRs, and lines added consume longer lead-times.

Table 2 presents the results of the regression using formula (2).

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	14.04	0.04	350.87	0.00
AIM	-0.65	0.09	-7.64	0.00

Table 2: Feature lead-time regression, AIM impact. 2908 features,$R^2 = .02$.

The R^2 value in Table 2 is very low due to large variability in the size of a feature. The estimate indicates that lead times for features not using AIM are 92% longer.

Feature defects

The response variable is a binary indicator on whether the feature had any field problem related MRs. The logistic regression formulas were as follows:

$$E(P(Fault)) = \frac{1}{1+e^{-\alpha-\beta_1 \log NDev - \beta_2 \log NMR - \beta_3 \log NLOC - \theta_{VE}}} \qquad (3)$$

$$E(P(Fault)) = \frac{1}{1+e^{-\alpha-\theta_{AIM}}} \qquad (4)$$

Table 3 shows the result of regression using formula (3).

	Estimate	Std. Error	z value	Pr(>|z|)
(Intercept)	-2.53	0.12	-20.61	0.00
Log(NDEV)	1.01	0.07	15.11	0.00
Log(NMR)	0.50	0.05	10.92	0.00
log(NLOC)	-0.30	0.02	-12.08	0.00
VE	-0.23	0.09	-2.67	0.01

Table 3: Feature quality logistic regression. VE impact. 15953 features, null deviance 9778, residual deviance 7812.

These estimates indicate that, for features with median number of logins (3), median number of MRs (3), and median number of lines (725), the probability of having field faults was 25% higher when VE was not used. While, as expected, features with more developers and MRs have an increased probability of having field faults, the number of lines (after adjusting for other factors) appears to decrease that probability. The large number of lines may be an indication of features that are implemented mostly outside the legacy code base where changes are easier to make and, therefore, more code is typically added.

	Estimate	Std. Error	z value	Pr(>|z|)
(Intercept)	-3.08	0.10	-30.08	0.00
AIM	-0.72	0.29	-2.50	0.01

Table 4: Feature quality logistic regression. AIM impact.

2908 features, null deviance 962, residual deviance 955.

To interpret the estimate, the features not using AIM were twice as likely to have a fault.

Impact on change properties

The introduction of AIM was believed to have another value-affecting impact: the reduction of developers. We investigate this hypothesis in this section. The regression formula is:

$$E(\log NDEV) = \alpha + \beta_1 \log NMR + \beta_2 \log NLOC + \theta_{AIM} \qquad (5)$$

Only the number of files had correlation less than .8 with the number of MRs for the AIM related features.

The results are in Table 5.

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	-0.01	0.01	-0.97	0.33
log(NMR)	0.46	0.01	39.88	0.00
log(NFILE)	0.10	0.01	15.62	0.00
AIM	-0.10	0.02	-6.02	0.00

Table 5: Number of developers in a feature, AIM impact. 2908 features, $R^2 = .59$.

We can see that even adjusting for the size of feature, the usage of AIM does appear to significantly decrease the number of developers involved in a feature. Thus, the technology enabled the production of features with fewer developers.

Related Work

The framework to evaluate the effects of a tool on development effort is described in (Atkins, et al., 2002). The methodology to assess the impact of Domain Engineering application environments is given in (Siy and Mockus, 1999). In this paper we extend and unify both frameworks to create a general approach for evaluating the impact of any software technology on lead-time and quality. We focus on practical applications of the approach by performing a detailed step-by-step analysis of two types of new technology.

This technique is very different in approach and purpose from other quality estimation techniques (such as COQUALMO (Chulani, 1999)), which make use of algorithmic or experiential models to estimate total project defects. Our approach is to estimate impact after actual development work has been done, using data primarily from change management systems. In addition, our approach is well-suited for quantifying the impact of introducing new technology to existing development processes.

We have previously investigated effects of these two technologies on effort spent on individual changes (Atkins, et al., 2000, Atkins, et al., 2002). Here we focus on modeling the lead-time and quality impact on software features, the units that provide added value to software by providing additional functionality.

Discussion

We present a methodology to quantify the impact from use of a software technology exemplified by a case study of a tool and an application engineering environment. We calculate the beneficial effects on the development of features, units that add business value. We find that by not using VE the lead-time increased by approximately 10% and the probability of field defect in a typical change increased by 25%. This is consistent with the design goals of the tool to make code more clear by hiding irrelevant code.

The use of the AIM application engineering environment resulted in halving the probability of a field defect in a feature. It also roughly halved the lead-time of the feature. Furthermore, the use of the environment was associated with the reduction of the number of people that work on a feature, consistent with previous results indicating significant effort savings and with the design goals of the technology.

Presently, the impacts are quantified in terms of reduction in the lead time and the probability of finding field faults. It would be useful to calculate the return-on-investment from introducing such technologies. We cannot obtain revenue data from features due to its proprietary nature, but we can estimate the savings to the organization. Reduction in lead time translates to savings in staffing costs due to the need for fewer developers and the expectation of freeing them up sooner to work on other features. Reduction in the probability of finding field faults translates to savings from fixing fewer faults. These savings offset the investment cost of introducing new technologies into the development process, and will be quantified in future work.

The described methodology is based on automatically extractable measures of software changes and should be easily applicable to other software projects that use source code version control systems. Since most of the change measures are kept in any version control system, there is no need to collect additional data.

This methodology is subject to a few limitations. Data to assess the impact of technological changes is only available after a few years of usage. It is also difficult to identify predictors that leave little if no imprint in the change database, for instance, technologies aimed at improving software testing.

We described in detail all steps of the methodology to encourage replication. We expect that this methodology will lead to more widespread quantitative assessment of software productivity improvement techniques. We believe that most software practitioners will save substantial effort from trials and usage of ineffective technology, once they have the ability to screen new technologies based on a quantitative evaluation of their use on other projects. Tool developers and other proponents of new (and existing) technology should be responsible for performing such quantitative evaluation. It will ultimately benefit software practitioners who will be able to evaluate appropriate productivity improvement techniques based on quantitative information.

Acknowledgements

For all statistical modeling and graphics we used the R package that is maintained and enhanced by a large group of volunteers worldwide. We also thank the anonymous reviewers for their helpful comments.

References

(Ardis and Green, 1998) M. A. Ardis and J. A. Green. Successful introduction of domain engineering into software development. *Bell Labs Technical Journal*, 3(3):10-20, September 1998.

(Atkins, et al., 2002) D. Atkins, T. Ball, T. Graves, and A. Mockus. Using version control data to evaluate the impact of software tools: A case study of the version editor. *IEEE Transactions on Software Engineering*, 28(7):625-637, July 2002.

(Atkins, et al., 2000) D. Atkins, A. Mockus, and H. Siy. Measuring technology effects on software change cost. *Bell Labs Technical Journal*, 5(2):7-18, April-June 2000.

(Atkins, 1998) D. L. Atkins. Version sensitive editing: Change history as a programming tool. In *Proceedings of the 8th Conference on Software Configuration Management (SCM-8)*, pages 146-157. Springer-Verlag, LNCS 1439, 1998.

(Boehm, 2003) Barry Boehm. Value-based software engineering. *ACM SIGSOFT Software Engineering Notes*, March 2003.

(Chulani, 1999) Sunita Chulani. Coqualmo (constructive quality model) a software defect density prediction model. *Project Control for Software Quality*, 1999.

(Coplien, et al., 1998) J. Coplien, D. Hoffman, and D. Weiss. Commonality and variability in software engineering. *IEEE Software*, 15(6):37-45, November 1998.

(Coplien, et al., 1987) J. O. Coplien, D. L DeBruler, and M. B. Thompson. The delta system: A nontraditional approach to software version management. In *International Switching Symposium*, March 1987.

(Cuka and Weiss, 1998) D.A. Cuka and D.M. Weiss. Engineering domains: executable commands as an example. In *Proc. 5th Intl. Conf. on Software Reuse*, pages 26-34, Victoria, Canada, June 2-6 1998.

(Herbsleb and Mockus, 2003) J. D. Herbsleb and A. Mockus. An empirical study of speed and communication in globally-distributed software development. *IEEE Transactions on Software Engineering*, 29(6):481-494, June 2003.

(Midha, 1997) A. K. Midha. Software configuration management for the 21st century. *Bell Labs Technical Journal*, 2(1), Winter 1997.

(Mockus and Votta, 2000) Audris Mockus and Lawrence G. Votta. Identifying reasons for software change using historic databases. In *International Conference on Software Maintenance*, pages 120-130, San Jose, California, October 11-14 2000.

(Mockus and Weiss, 2000) Audris Mockus and David M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169-180, April-June 2000.

(Mockus, et al., 2003) Audris Mockus, David M. Weiss, and Ping Zhang. Understanding and predicting effort in software projects. In *2003 International Conference on Software Engineering*, pages 274-284, Portland, Oregon, May 3-10 2003. ACM Press.

(Pal and Thompson, 1989) A. Pal and M. Thompson. An advanced interface to a switching software version management system. *In Seventh International Conference on Software Engineering for Telecommunications Switching Systems*, July 1989.

(Perry, et al., 2001) D. Perry, H. Siy and L. Votta. Parallel Changes in Large Scale Software Development: An Observational Case Study. *ACM Transactions on Software Engineering and Methodology*, 10(3):308-337, July 2001.

(Rochkind, 1975) M.J. Rochkind. The source code control system. *IEEE Trans. on Software Engineering*, 1(4):364-370, 1975.

(Rogers, 1995) E. M. Rogers. *Diffusion of Innovation*. Free Press, New York, 1995.

(Siy and Mockus, 1999) H. Siy and A. Mockus. Measuring domain engineering effects on software coding cost. In *Metrics 99: Sixth International Symposium on Software Metrics*, pages 304-311, Boca Raton, Florida, November 1999.

(Weiss and Lai, 1999) D. Weiss and R. Lai. *Software Product Line Engineering: A Family-Based Software Development Process*. Addison-Wesley, 1999.

Author Biographies

David Atkins is an assistant professor in Computer Science at the American University in Cairo. He came to Egypt from the University of Oregon, and for most of his career, he was a member of technical staff in the Software Production Research Department at Bell Labs in Naperville, Illinois. His research interests include programming languages and software version management. He received a B.A. in mathematics from the College of Wooster in Ohio and a Ph.D. in mathematics from the University of Kansas in Lawrence.

Audris Mockus conducts research on quantifying, modeling, and improving software development. He designs data mining methods to summarize and augment software change data, interactive visualization techniques to inspect, present, and control the development process, and statistical models and optimization techniques to understand the relationships between people, organization, and characteristics of a software product. Audris Mockus received B.S. and M.S. in Applied Mathematics from Moscow Institute of Physics and Technology in 1988. In 1991 he received M.S. and in 1994 he received Ph.D. in Statistics from Carnegie Mellon University. He works in the Software Technology Research Department of Avaya Labs. Previously he worked in the Software Production Research Department of Bell Labs.

Harvey Siy received the B.S. degree in Computer Science from University of the Philippines in 1989, and the M.S. and Ph.D. degrees in Computer Science from University of Maryland at College Park in 1994 and 1996, respectively. He is a Member of Technical Staff at Lucent Technologies doing capacity and performance engineering for the 5ESS product. He was previously with the Software Production Research Department of Bell Labs, where he conducted empirical studies of large scale, industrial software engineering processes.