AI-Based Software Defect Predictors: Applications and Benefits in a Case Study

Ayse Tosun Misirli, Ayse Bener, and Resat Kale

■ *Software defect prediction aims to reduce software testing efforts by guiding testers through the defect-prone sections of software systems. Defect predictors are widely used in organizations to predict defects in order to save time and effort as an alternative to other techniques such as manual code reviews. The usage of a defect prediction model in a real-life setting is difficult because it requires software metrics and defect data from past projects to predict the defect-proneness of new projects. It is, on the other hand, very practical because it is easy to apply, can detect defects using less time, and reduces the testing effort. We have built a learning-based defect prediction model for a telecommunications company in the space of one year. In this study, we have briefly explained our model, presented its payoff, and described how we have implemented the model in the company. Furthermore, we compared the performance of our model with that of another testing strategy applied in a pilot project that implemented a new process called team software process (TSP). Our results show that defect predictors can predict 87 percent of code defects, decrease inspection efforts by 72 percent, and hence reduce postrelease defects by 44 percent. Furthermore, they can be used as complementary tools for a new process implementation whose effects on testing activities are limited.*

Software defects are more costly if discovered and fixed in the later stages of the development life cycle versus during production (Brooks 1995). Therefore, testing is one of the most critical and time-consuming phases of the software development life cycle, which accounts for 50 percent of the total cost of development (Brooks 1995).

The testing phase should be planned carefully in order to save time and effort while detecting as many defects as possible. Different verification, validation, and testing strategies have been proposed so far to optimize the time and effort utilized during the testing phase: code reviews (Adrian, Branstad, and Cherniavsky 1982; Shull et al. 2002), inspections (Fagan 1976), and automated tools (Menzies, Greenwald, and Frank 2007; Nagappan, Ball, and Murphy 2006; Ostrand, Weyuker, and Bell 2005). Defect predictors improve the efficiency of the testing phase in addition to helping developers assess the quality and defect-proneness of their software product (Fenton and Neil 1999). They also help managers in allocating resources. Most defect prediction models combine well-known methodologies and algorithms such as statistical techniques (Nagappan, Ball, and Murphy 2006; Ostrand, Weyuker, and Bell 2005; Zimmermann et al. 2004) and machine learning (Munson and Khoshgoftaar 1992; Fenton and Neil 1999; Lessmann et al. 2008; Moser, Pedrycz, and Succi 2008). They require historical data in terms of software metrics and actual defect rates, and combine these metrics and defect information as training data to learn which modules seem to be defect prone. Based on the knowledge from training data and software metrics acquired from a recently completed project, such tools can estimate defect-prone modules of that project.

Copyright © 2011, Association for the Advancement of Artificial Intelligence. All rights reserved. ISSN 0738-4602

Recent research on software defect prediction shows that AI-based defect predictors can detect 70 percent of all defects in a software system on average (Menzies, Greenwald, and Frank 2007), while manual code reviews can detect between 35 to 60 percent of defects (Shull et al. 2002) and inspections can detect 30 percent of defects at the most (Fagan 1976). Furthermore, code reviews are labor intensive since, depending on the review procedure, they require 8 to 20 lines of code (LOC) per minutes for each person in the software team to inspect the source code (Menzies, Greenwald, and Frank 2007). Therefore, AI-based models are popularly used by various organizations (Menzies, Greenwald, and Frank 2007; NASA MDP 2007; Nagappan, Ball, and Murphy 2006; Nagappan, Murphy, and Basili 2008; Ostrand, Weyuker, and Bell 2005). These predictors learn specific patterns concerning defect-proneness from past projects and use this information to predict the defect-proneness of new projects. As more projects are observed throughout the development life cycle, more data is collected, and predictions are more accurate.

We conducted a comprehensive metrics program and built a defect prediction model at a large telecommunications company in Turkey during a period of one year (Tosun, Bener, and Turhan 2009). During this metrics program, we collected static code metrics and churn metrics from the company's 9 projects in 10 releases. We matched the prerelease defects (the defects detected during the testing phase) of the previous releases with the source code at file level. Then we made predictions on the new releases of the projects. We calibrated our model based on its prediction accuracy and discovered that it is possible to detect on average 88 percent of defective files using a defect predictor (Tosun, Turhan, Bener 2009). We also compared our model in terms of the gain in inspection effort against a random testing strategy. It is seen that our defect predictor can reduce inspection efforts by 72 percent.

In this article, we describe our model from a machine-learning perspective with measurable benefits such as the defect detection capability and improvements in the cost-benefit analysis. We present the payoff of the model used and show how the model has been implemented in the company. We also compare the model's performance with a pilot process transformation, which applies labor-intensive checklists and formal procedures to detect defects during testing. In terms of time and effort spent for finding defects, manual code reviews, inspections, and unit testing constitute 25 percent of the total effort. Our results show that our prediction model automatically finds 75 percent of the defects detected in unit testing, code reviews, and inspections only in a few seconds. Therefore, we conclude that defect predictors can augment process changes during testing activities by lowering defects and inspection efforts.

Brief Information about the Organization

The organization we collaborated with in this study is the leading telecommunications (Global System for Mobile Communications, or GSM) operator in Turkey and the third biggest operator in Europe in terms of number of subscribers. As of 31 December 2009, it is providing mobile communication services to 35.4 million subscribers, with an additional 26.1 million subscribers in Azerbaijan, Kazakhstan, Georgia, Ukraine. and Northern Cyprus. It was founded in 1994, and since 2006, it has had a research and development (R&D) center with around 200 engineers. In this R&D center, the company develops software products and solutions for mobile operators all over the world. Some of these solutions are network solutions, value-added services, subscriber identity module (SIM card)–related solutions, terminal-based solutions, billing and charging solutions, data mining, data warehouse, and customer and channel management systems and applications. Its legacy system contains millions of lines of code that are being maintained. The majority of its software is implemented with Java, JavaServer Pages (JSP), PL/SQL, and other new technologies such as service-oriented architecture (SOA).

As with any other company, time and budget constraints put constant pressure on R&D. As customers require new functionality or technology changes, the company has to respond faster and faster with new software releases. Therefore, its approach to development is incremental, with each new release adding new functionality or a software modification to previous releases. In such a limited time, the software team cannot apply any measurement process to assess the overall software quality. Therefore, there was an urgent need to implement a measurement and analysis program to monitor defects, reduce defect rates and testing effort, and to improve software quality. We have built a measurement repository, bug tracing and matching system, and a defect prediction model for the company. In this study, we explain the implementation of the defect prediction model after it has been calibrated with local data to achieve the highest prediction accuracy.

Description of the Prediction Model

Our learning-based defect predictor is a typical machine-learning application: it contains a train-

Attribute	Description	Attribute	Description
Cyclomatic density, vd(G)	The ratio of the module's cyclomatic complexity to its length	Essential complexity, ev(G)	The degree to which a module contains unstructured constructs
Design density, dd(G)	condition/decision	Cyclomatic complexity, v(G)	Number of linearly independent paths
Essential density, ed(G)	(ev(G) – 1) / (v(G) – 1)	Maintenance severity	ev(G)/v(G)
Difficulty (D)	1 / L	Length (N)	$N1 + N2$
Level (L)	(2 / n1) * (n2 / N2)	Programming effort (E)	$D * V$
Volume (V)	N*log(n)	Programming time (T)	E / 18
Unique operands	n1	Executable LOC	Source lines only with code and white space
Branch count	Number of branches	Total operators	N1
Decision count	Number of decision points		Total operands
Condition count	Number of conditionals	Unique operators	n2

Table 1. List of Static Code Attributes (NASA 2007)

ing phase to learn from the data related to previous projects and a testing phase to predict the potential defect-free and defective modules of the new project. A module could be a package, class, file, or method inside the source code. Erroneous predictions of the defect-free modules in the form of defects (false alarms) force testers to inspect "safe" modules and waste their precious time. On the other hand, missing defective modules (false negatives) would cause more expensive and hard-to-fix failures on the final software product. Thus, false negatives need to be avoided.

Basic Terminology: Input and Output Variables

As a classification task, our input variables are a set of static code attributes, such as lines of code, complexity, and operand and operator counts, extracted from the source code. Static code attributes are widely used and easily collected through automated tools (Menzies, Greenwald, and Frank 2007; Moser, Pedrycz, and Succi 2008; Lessmann et al. 2008) and proposed by various researchers such as McCabe (1976) and Halstead (1977). The full list of attributes collected from the source code in this study is illustrated in table 1.

In the literature, various researchers have also used other types of metrics such as object-oriented design metrics (Basili, Briand, and Melo 1996; Chidamber and Kemerer 1994), in-process metrics (Nagappan, Ball, and Murphy 2006), and organizational metrics (Nagappan, Murphy, and Basili 2008) in order to predict defects. Although increasing the information content of input data by adding different types of metrics has positive effects on defect prediction capability, it is not easy to collect in-process and organizational metrics from an organization. Therefore, we have preferred to rely on the source code as a basis for collecting metrics, that is, input variables.

In addition to code attributes, there are class labels for each module such as 0 as defect free and 1 as defective in the training set. If a module in the software system has been associated with a bug (code defect) during the testing phase, it is labeled as 1; otherwise, it is labeled as 0. It is not necessary to count the number of defects a module is associated with since our aim in this study is not predicting the number of defects. More precisely, the training set is an N-by-M matrix where N is the number of modules taken from past projects and M is the number of code attributes (M-1) extracted from their source code as well as a class label to indicate whether a defect has been detected on that module during testing.

The test set, on the other hand, contains attributes extracted from the modules of a new project whose defect labels are unknown. Therefore, the output variable (Y) of the model would be the class labels of modules in the test set as defect free or defect prone.

The Use of AI Technology

We have used a naïve Bayes classifier as the algorithm of our prediction model. The Bayes theorem defines the posterior probability as proportional to the prior probability of the class $p(C_i)$, and the likelihood of attributes, $p(X|Y = C_i)$ with strong independence assumptions on attributes (see Alpaydin [2004]). In binary classification problems such as defect prediction, naïve Bayes computes the posterior probability of a module being defective, or the probability of a module being defect free, given its attributes. Then, it assigns a module to the defective class if its posterior probability is greater than a predefined threshold (0.5). Otherwise, the module is classified as defect free.

We have used a naïve Bayes classifier for several reasons. First of all, it is a widely used, simple, and robust machine-learning technique in various

Actual	Predicted Defective	Predicted Defect Free
Defective	TP	FN
Defect free	FP	TN

Table 2. Confusion Matrix.

applications such as pattern recognition (Kuncheva 2006), medical diagnosis (Uyar et al. 2009), and defect prediction (Menzies, Greenwald, and Frank 2007; Moser, Pedrycz, and Succi 2008; Tosun, Bener, and Turhan 2009). It is also easy for field practitioners to understand and implement. Second, defect prediction models with a naïve Bayes classifier deliver the best prediction accuracy on public datasets compared with models with other classifiers (Menzies, Greenwald, and Frank 2007). One of the reasons for the success of the naïve Bayes classifier over other methods is that it combines signals coming from multiple sources. It is not affected by the "brittleness" of data (minor changes in training sample do not give completely different results), since it polls numerous Gaussian approximations to the numeric distributions (Menzies, Greenwald, and Frank 2007). Therefore, minor correlations between attributes or samples in the training set within the field of software defect prediction do not confuse naïve Bayes classifiers. Third, a recent study by Lessmann et al. (2008) finds that the importance of classification algorithms in defect prediction may be less than previously assumed, since no significant performance differences exist among the top 17 classifiers. This result is very important for our case study since it reduces the necessity of trying all classification techniques. Thus, instead of applying different algorithms, we have selected naïve Bayes as the algorithm of our model and focused on calibration based on local data.

Performance Evaluation

We use receiver operator characteristics (ROC) curves to assess the discriminative performance of a binary naïve Bayes classifier.[1] In a ROC curve, our objective is to reach the point (1, 0) in terms of (y, x), where the y-axis represents the true positive rate and the x-axis represents the false positive rate. We have computed these performance measures to evaluate the accuracy of our model. However, similar to defect prediction research (Menzies, Greenwald, and Frank 2007; Lessmann et al. 2008), we name the true positive rate as the probability of detection rate (*pd*) and the false positive rate as the probability of false alarm rate (*pf*) in this study. The ideal classification, point (1, 0) in a ROC curve, can be reached when we correctly classify all defective modules (*pd* = 1, that is, 100 percent) with no false alarms (*pf* = 0, that is, 0 percent).

Finally, the common performance measures are derived from a confusion matrix (table 2) where {*TP, FP, FN, TN*} are {true positive, false positive, false negative, and true negative} rates respectively (Menzies, Greenwald, and Frank 2007). Probability of the detection rate (*pd*) is a measure of accuracy for correctly classifying defective modules. It corresponds to the true positive rate in machine learning and should be as close to 1 as possible: $(pd) = TP / (TP + FN)$.

Probability of the false alarm rate (*pf*) is a measure of accuracy to represent the false alarms when we misclassify defect-free modules. We must avoid high *pf* rates in software defect prediction models since they would increase the testing effort: $(pf) = FP / (FP + TN)$.

It is very rare to achieve the ideal case with 100 percent *pd* and 0 percent *pf* rates using a prediction model. When the model is triggered often to increase the *pd* rate, the *pf* rate would, in turn, increase. Therefore, our objective is to get as high *pd* rates as possible while keeping *pf* rates at a minimum.

Utilization of the Model and Payoff

We built our defect prediction model on the software system of a telecommunications company. Previously, a tool that collected metrics from the source code did not exist. Moreover, although the defects were logged in a version management system, they were not matched with the source code at any granularity level, that is, package, file, method, or LOC.

We started a metrics program to collect the required data for building our defect predictor. We developed an open-source metrics extraction and a defect prediction tool called Prest (Kocaguneli et al. 2009) and collected code metrics from Java and JSP files. Previously, there was no process in the company for bug tracing. Defects were not often stored during development activities. Furthermore, there was no process to match defects with the files in order to keep track of the reasons for any change in the software system. Therefore, we implemented an organizationwide process change that is fully supported by the senior management (Tosun, Benar, and Turhan 2009). This process change helped us to store defects as well as to match them at the file level.

Static code attributes at the file level and defect labels matching the files (more precisely, Java and JSP files) were collected from 9 different projects in 10 releases, and this dataset was donated to a pub-

lic data repository, Promise.[2] Then, the project-based defect prediction was performed such that the defective files of a project at release *n* were predicted using the static code attributes and the defect labels of the same project at release *n*-1. This methodology was used in the literature before (Ostrand, Weyuker, and Bell 2005) and is well suited in our case since each release was treated as a new project. Based on this training-testing strategy, we assessed the performance of our predictor and discovered that the deployed model with a naïve Bayes classifier correctly classifies 90 percent of the defective files while producing 50 percent false alarms (Tosun, Bener, and Turhan 2009). Since false alarms were very high, we included a new software metric, the version history flag, in addition to static code attributes, to indicate the latest activity date on files as inputs to the model. This flag shows whether a file has been edited at least once in six months. If a file does not have any activity for a long time, then it is less likely that the file contains defects. Using version flags further improved the prediction performance by decreasing the false alarm rates on an average of 28 percent, from 50 percent to 22 percent (Tosun, Bener, and Turhan 2009).

Table 3 shows the summary of the prediction performances in nine releases. We have made predictions for an average of three projects in every release and took the mean and the standard deviation of the prediction performances in terms of *pd* and *pf* rates. As seen in table 3, we have successfully achieved an 87 percent detection rate in nine releases with 26 percent false alarms. Our defect predictor helps in detecting defective modules using less time and effort. Furthermore, it guides testers through specific files and reduces the inspection effort compared to code reviews and inspections. The process is less labor intensive if local data is collected as required.

Releases	*pd*	*pf*	GE
2	77	33	58
3	92	21	81
4	82	23	78
5	75	15	74
6	87	18	83
7	83	21	71
8	98	33	68
9	88	29	72
10	97	41	68
Average	87	26	72.5
(Standard Deviation)	(8.1)	(8.5)	(7.6)

Table 3. Performance of the Prediction Model.

The practical benefits of using a defect predictor have been further computed using a cost-benefit analysis from Arisholm and Briand (2006). The authors compared the inspection effort suggested by a defect prediction with a random testing strategy. Based on that, the gain in the effort (GE) can be calculated with the following formula: $100 \times ((\text{MRT} - \text{MDF}) / \text{MRT})$.

In this formula, MRT represents the number of modules (files in our study) that must be inspected through a random testing strategy, whereas MDF represents the number of modules that must be inspected with a defect predictor. We have conducted the cost-benefit analysis of our predictor to present the practical benefits for the company. If we would use a random testing strategy, we would have to inspect 87 percent of the files (12,750 LOC out of 15,000 LOC per release on average) to be able to detect 87 percent of defects. However, our model highlights only 25 percent of files that contain 88 percent of defects. Therefore, the gain in the inspection effort is 72 percent. Table 5 shows that the implemented model reduces the inspection effort by 72.5 percent on average through highlighting the critical parts in the software system. Rather than looking at 87 percent of the files, we can inspect only 24 percent of the files (3750 LOC out of 15,000 LOC per release on average) and detect 87 percent of defects.

Deployment of the Model

The prediction results given above were so satisfactory that the quality assurance team at the company decided to integrate the model into the company's configuration management system. It planned to use the prediction model prior to the testing phase so that the defect-prone files would be investigated by either the developer before he/she transfers the project to the test team or the tester so that his/her effort would be assigned to the critical parts only. As mentioned previously, we have implemented Prest (Kocaguneli et al. 2009), an open-source metrics extraction and defect prediction tool, during this study. This tool not only extracts code metrics from different granularity levels of projects written in Java, JSP, C, and C++, but also includes a defect prediction component in which a naïve Bayes classifier can be executed on a new project given a training set.

We have customized the defect prediction component of Prest for the company. A graduate student from our research laboratory and an engineer from the company completed the implementation of this tool on the company's configuration management system. A Java program was implemented to perform the following steps: (1) We wrote a shell script to call Prest and extract code metrics from a

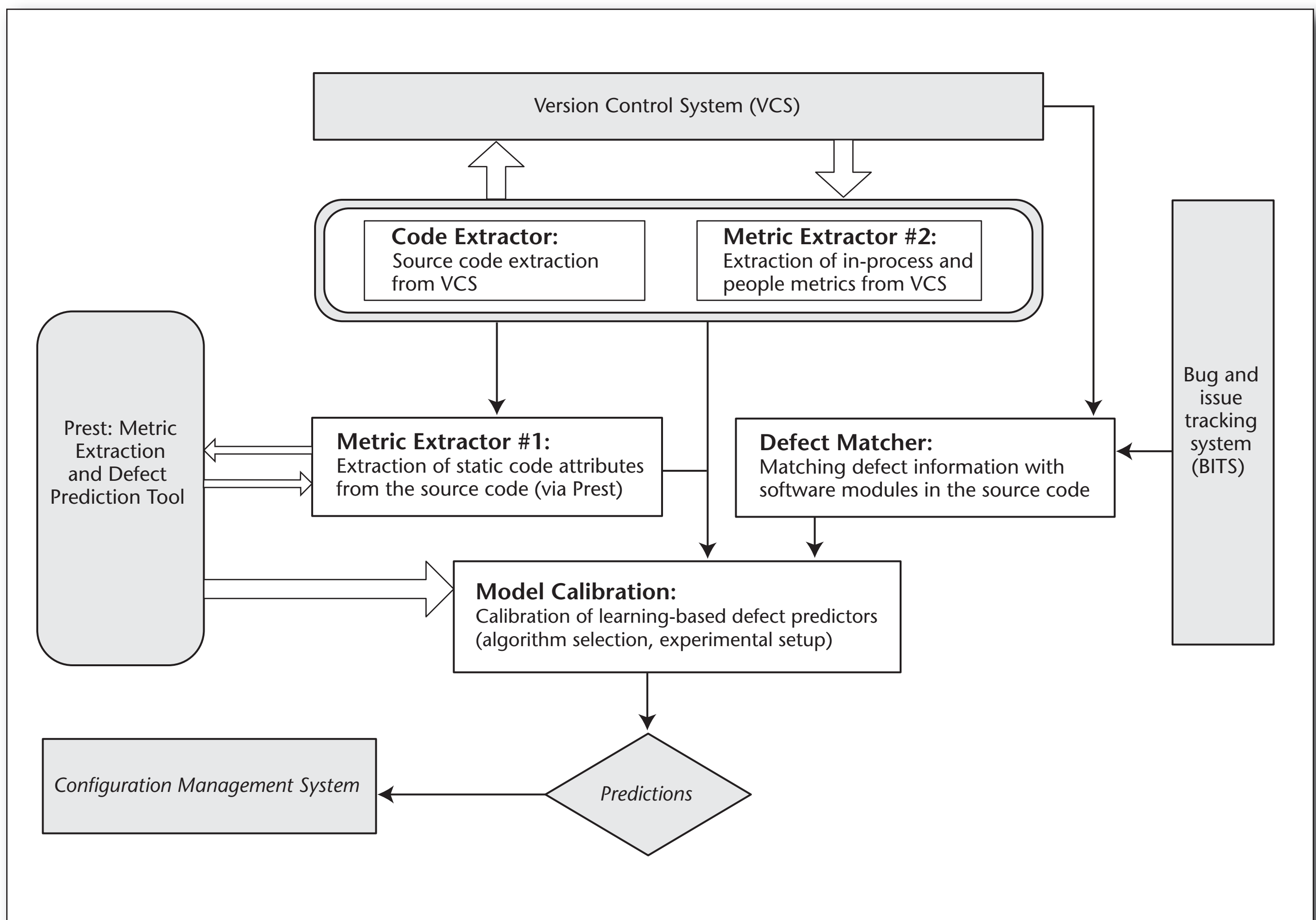

Figure 1. Architecture of the Deployed Prediction Model.

specified release of a specified project. (2) Shell scripts were to retrieve the defects detected for a specified project from the bug-tracking system and match those defects with files that already included code metrics. (3) We prepared the training set from the previous release of a specified project. (4) We prepared the test set from the current release of a specified project. (5) We called Prest once again to activate the prediction component, load the training and test sets, and run the algorithm and make the prediction. (6) The program returned defect-prone files of a specified project for its current release.

In figure 1, the architectural schema of deployment of the prediction model is presented. There are five major modules: code extractor extracts a stable version of the source code for a specified project from the company's version control system (VCS). Then, metric extractor 1 is run to trigger Prest for extracting code metrics from the given source code version. Metric extractor 2 mines the bug and issue tracking systems (BITS) as well as VCS in order to extract in-process and people metrics, such as commits, edited LOC, unique committers. Once metrics are ready for training or test sets, the defect matcher retrieves bug reports from BITS to match the bugs with associated modules. Finally, model calibration calls Prest once more to build the defect prediction. This module has an AI component, in which a training set is used to find the parameters of the selected algorithm and a test set is used to make predictions.

We formed a more generic system during deployment such that metric extractors 1 and 2 can be easily updated with new scripts if VCS or BITS is changed. Then, the prediction model would still be used within the organization.

The output of the model (see the predictions node in figure 1) is fed into the configuration management system (CMS). The CMS lists all source files of a specified project with their code metrics as well as predictions on their defect-proneness on a single web page.

Based on feedback from the software team in the

organization, we updated the output of the prediction model. In figure 1, the output variable is not only a class label (for example, 1: defect prone or 0: defect free), but also a probability for defect-proneness (for example, defect prone with 75 percent probability), since the latter helps the software team decide "where to start." Testers would look at files with the highest probability of defect-proneness when they start testing the system.

The local prediction model has been used on two major components of the software system for six months. It lists defect-prone files of these components at the beginning of the testing phase so that testers' efforts would be assigned to critical parts. Every 2 weeks, a new release with 10 to 15 work packages and more than 400 interfaces of these components is being published. Since the release period is short, each release package contains at most a functionality or two new functionalities and the rest is modifications/upgrades for the current system. These work packages are tested using 1000 to 1500 test cases by a total of 20 testers. Due to time constraints, the testing phase is limited to five days on average. Thus, each tester needs to run 10 to 15 automated test cases per day (in eight hours) in order to inspect 80 percent of the functionality in total. It is also necessary for each tester to conduct manual inspections to ensure 100 percent test coverage.

On the other hand, in reality, the development phase is delayed with frequent requirement changes due to revisions in government regulations. Thus, testers often have only three days to complete the verification of a release. During this period of time, a tester can execute 30 to 45 test cases. All test cases executed by 20 testers in three days can cover only 48 percent of the functionality. Therefore, the company applied our defect prediction model to prioritize critical parts of the code and assign the company's few resources to those parts immediately. The model inspects 24 percent of the files corresponding to 35 (23 percent) different functionalities, and it detects 87 percent of defects. As a result, each tester is required to run 9 to 13 automated test cases per day to inspect 71 percent of the functionality in total. In other words, the company has managed to decrease the effort in person-hours per defect from 1.25 to 1.1 (a decrease of 11.2 percent) with the help of our defect prediction model. The quality assurance team also counted the number of postrelease defects for the last five releases and found that, since the model successfully catches most of these defects during the testing phase, postrelease failures due to a code defect have been decreased from 59 percent to 32 percent (a decrease of 44 percent). Table 4 also summarizes the benefits after using a defect prediction model (DPM) in terms of inspection effort, test cases, and postrelease defects.

Metric	Before DPM	After DPM
Number of executed test cases per day	10–15	9–13
Prerelease defects found during testing (percent)	N/A	87
The amount of functionality inspected (percent)	48	71
Inspection effort (person-hour)	1.25	1.1
Postrelease defects (percent)	59	32

Table 4. Deployment of the Model: Benefits.

Using Defect Prediction to Support a New Process Implementation

Software organizations have been using different development methodologies (such as agile development, capability maturity models, team software process) since the 1990s, in order to produce superior software systems in terms of improved code quality, reduced defect rates, and effective resource allocation.[3] Although these methodologies may seem different, they are most effective when they are used to complement each other since they fix problems in different phases of a software life cycle. Similarly, software defect predictors are useful tools to complement these methodologies to improve software quality and productivity to reduce defect rates.

We have done an additional analysis to examine whether a new process implementation on its own is successful at improving the quality of a software product, that is, decreasing the defect rates and reducing the testing effort. Recently, a pilot project has been conducted for implementing a new process, team software process (TSP), in the company.

We first investigated how TSP meets the expectations in terms of effort, schedule, and defect rates. To accomplish this, we observed estimated versus actual effort spent on each phase in the development life cycle. Second, we applied our defect prediction model on the pilot project after it was completed and analyzed whether the defect prediction model could be a complementary approach to achieve above and beyond the benefits provided by TSP, such as reduced defect rates and testing efforts.

What Is Team Software Process? Along with personal software process (PSP), TSP (see note 3) helps engineers ensure the quality of software products, create secure software products, and improve process management in an organization. Engineering groups often use TSP to apply integrated team concepts to the development of software-intensive systems (see note 3). A launch process leads the teams and managers to establish their goals, define the roles within the team, assess risks, and produce a team plan. This process first directs

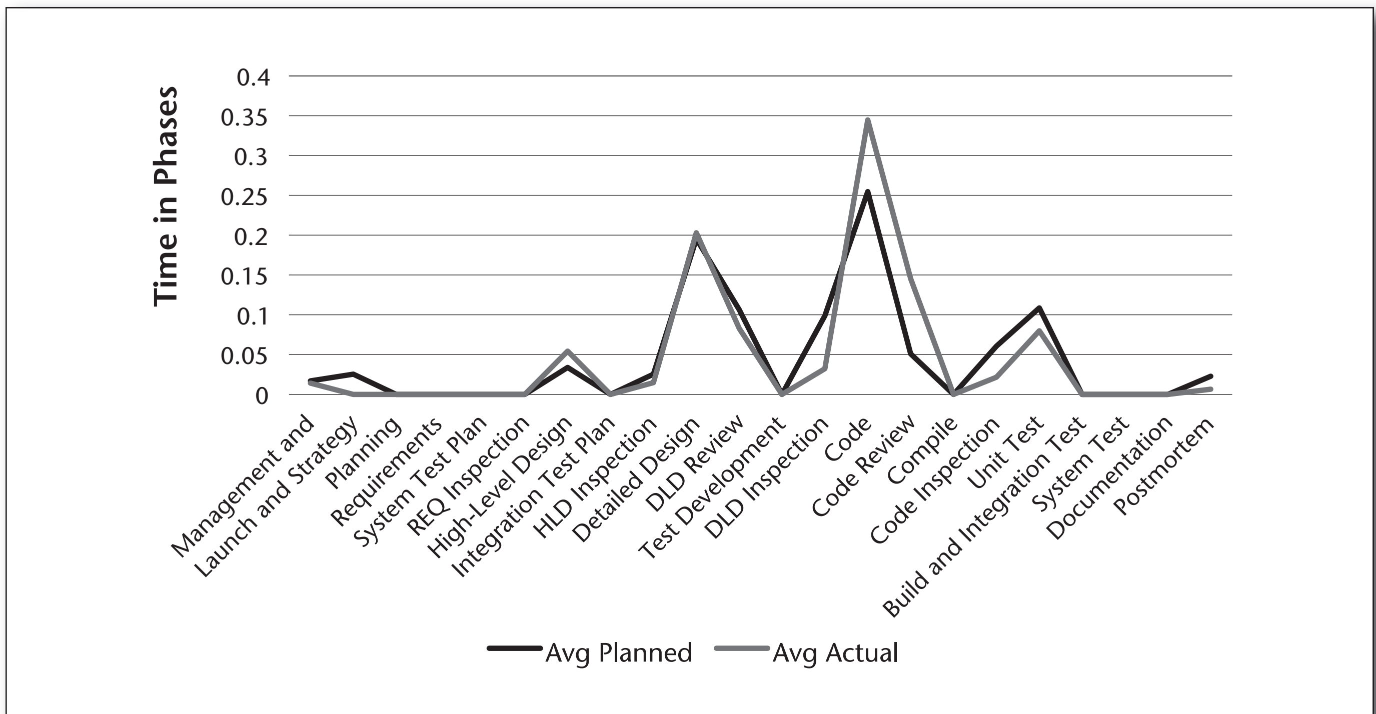

Figure 2. Time in Hours Spent for Each Phase.

the goals and the plans of engineers in the company individually. Then it helps create self-directed teams who take ownership of their plans and processes and direct their tasks accordingly. Using PSP, engineers do not only improve the process of planning and estimating the size and effort related to their tasks, but they also understand the means of managing quality and reducing defects. According to case studies carried out in various organizations such as Motorola, engineers have achieved less than 0.1 defects per thousand LOC on nearly 18 projects (see note 3). Although the objectives and claims of applying such a process are very strong, it is clear that TSP, along with PSP, obliges engineers individually to ensure that they adopt good practices in terms of engineering disciplines.

Analysis on the Pilot TSP Project. The management selected a software team of four people to develop a new project by applying fundamental principles of TSP. The software team was asked to report all tasks it accomplished and planned. It also kept actual time and effort required for completing these tasks and the number of defects detected and removed during testing activities.

The team's objectives were to observe the applicability of TSP in its organization and evaluate the benefits of the process change in terms of productivity, estimation accuracy, the defects detected in unit testing, and defect density in the testing and production phases.

The pilot project would provide a modification for one of the four major software components in the large software system. When compared with the large system, for which we have implemented a predictor model, the size of the pilot project can be viewed as one-fifth of the entire system. At the end of development life cycle, it contains 107 Java packages with around 105,925 executable LOC. The pilot project team executed two launches during the project life cycle. Every launch started by defining the tasks required for completing the project, assigning each task to an engineer (or a group of engineers), estimating the time and size of the tasks and the number of defects injected and removed for each phase.

Engineers in the pilot group purposefully applied the TSP principles for almost six months. They prepared reports with statistics on estimation accuracy, productivity, and defect rates. Based on these reports, we have formed a chart that represents the actual time spent in software phases (in percent) aligned with the estimated time periods in figure 2.

By looking at this figure, we can see that the team actually applied TSP and made accurate estimations in terms of the time spent in each phase. It spent 95 percent of its total effort on high-level design (HLD), detailed design (DLD), DLD review, implementation, code review, and unit tests. Specifically, coding and code review took slightly

more than expected. Code reviews, inspections, and review constitute 25 percent of this effort.

When we observe the defects detected and removed in each phase (table 5), we see an interesting pattern: 25 percent of all defects were detected and fixed during code reviews and inspections in comparison to 28 percent of defects detected during unit testing.

A panel on IEEE Metrics reports that code reviews and manual inspections can detect 60 percent of defects on average (Shull et al. 2002). However, although 17 percent of total time was spent for code reviews and inspections in the course of this pilot project, the percentage of the defects detected during independent testing activities was still 29 (that is, testing phase). TSP should ideally increase the number of defects detected in code reviews and unit testing since it provides a guide for increasing the quality of the work of engineers (software developers). Furthermore, TSP argues that it should decrease the defect density in the testing phase and increase the software quality.

Defect Prediction on the TSP Project. We argue that when a learning-based defect predictor was used as complementary to code reviews and inspections during the pilot TSP project, it would help reduce both the time and effort spent during coding and testing. To strengthen our claim, we have used the local prediction model on the pilot project data and identified defect-prone modules during "coding" (including code reviews, inspection, and unit testing). This was also an after-the-fact analysis of our prediction model that would show what such models could provide during a new process implementation.

In the pilot TSP project, defect logs were kept in detail, and when possible, they were matched with a Java package in the project. We used Prest to extract static code attributes from the Java packages of the pilot project. Then we matched the defects detected during "coding" with code metrics of the packages. We assigned 1 as defective and 0 as defect free to every package in the source code if there was a minimum of one defect. Finally, we made package-level predictions on the pilot project. Table 6 summarizes the defect ratio in the package level and the prediction performance of our predictor. It shows that using a smart and automated tool, we managed to detect 75 percent of all defects without spending too much effort on inspecting the entire code through the use of labor-intensive checklists. We can decrease the inspection effort by 10 percent compared to a random testing strategy. This gain in the inspection effort is lower than what we proposed in table 3 due to the granularity level we used in TSP analysis. Matching the files with defects rather than packages would prove to be more beneficial for reducing the inspection efforts. Thus, we see that a

Phase Removed	Count
Unit Testing	66
Code Inspection and Reviews	58
DLD Review	38
Design	6
Independent Testing	68
Total	236

Table 5. Number of Defects Detected and Fixed in Phases.

process change itself is limited to reduce the inspection effort or to improve the quality of coding in terms of the number of defects detected during unit testing.

If we used such a tool during the TSP implementation, it would enable us to save time and find the defects that were missed during the "coding" phase but detected during the testing phase. The rate of false alarms seems to be high—a fact that would waste the limited testing effort on actual defect-free modules. However, in this analysis, we only predicted the defects detected during the "coding" phase, but we did not match the defects detected during the testing phase with the software packages. Therefore, the packages that are misclassified as defect prone (false alarms) may also contain a defect detected during the testing phase.

To sum up, TSP aims to find more defects and improve software quality by guiding developers to do more code reviews and more inspections. Defect prediction also aims to find more defects by optimizing the inspection effort. In other words, an AI-based defect prediction model finds more defects with less inspection effort compared to manual code reviews. In our analysis we have seen that our model found more defects over and above the ones found during the TSP project implementation. Had the AI-based defect prediction model been used during TSP project, it would have decreased the manual inspection effort and found more defects. We have observed that process change is beneficial to increase planning accuracy and obtain high-quality software products. However, they may not provide a solution to all problems such as decreasing the defects, increasing quality, or accurate estimation of size and effort in the software development life cycle. Defect predictors can be used effectively to complement and further improve such process models.

Maintenance

Similar to many AI-based models, our model also requires calibration. The company decided to train the model with new data in periods of three

Number of Attributes	Number of Packages	Defectives (percent)	*pd*	*pf*	GE
20	107	15 percent	75 percent	26 percent	10 percent

Table 6. Predictions on the Pilot Project Using Defect Predictor.

months and make predictions on new releases. Since the model has been successfully integrated with the company's software system, it is just a few minutes to form a new training set (using Metric Extractor and Defect Matcher modules) by running automatic scripts every three months. The company also motivates the teams for making such tools part of their routine during the development and testing stages. This way, it will be easier to use the model in collaboration with the development and test teams in order to analyze the code quality and to predict the critical parts of the software. Using the available training set, the model would calculate the parameters and predict defect-prone files of a test set without human intervention. Furthermore, we plan to track the prediction performance and the usage of the model in the company for one year and will calibrate the algorithm if necessary. The overall maintenance effort is less than 30 minutes per month, and this shows that the cost of implementing the model is well worth it.

Predicting Final Reliability

Innovative applications of AI techniques have tangible benefits both for academia and industry. In terms of research opportunities, this project helps us focus on a new challenge in software engineering: when to stop testing and release the software. Building prediction models is quite effective in estimating the defect-proneness of software systems before production. They can also be used to estimate postrelease failures (Nagappan, Ball, and Murphy 2006; Nagappan, Murphy, and Basili 2008) and fix them prior to the release. However, it is still an issue in practice to decide when to release the software. Software managers in large organizations look for more extensive models to answer the questions of "when to stop testing" and "how much reliability will improve with more testing." In real life, as software systems get more complex, software managers monitor various software factors related to different phases in the development life cycle. They combine their prior knowledge with these facts in order to ensure that the software has reached a predefined reliability level and it is ready to be deployed.

As a new research project, we built a comprehensive Bayesian network (BN) that would improve decision making for project managers by estimating a reliability (confidence) level for the software before the production phase. We chose Bayesian networks for various reasons. First, BNs allow learning causal relations between software factors (Heckerman 1995), second, BNs combine data and prior expert knowledge using statistical techniques (Heckerman 1995), and finally, BNs let users observe the effects of one variable on another or on the final node, and hence, they provide efficient trade-off analysis for software managers. The proposed BN includes subnetworks such as requirements analysis, design, development, testing, and project management. Each subnet consists of various software artifacts whose causal relationships would represent the reliability of the corresponding process. Combination of these subnets would estimate the final reliability of a software system.

We collected process metrics and postrelease defect counts from 10 releases of the software system in the company and predicted their number of critical postrelease defects (a range is predicted rather than a single number) that would probably occur after the release. Based on the predefined confidence ranges, we estimated the confidence levels of these releases. The actual number of postrelease defects found after these 10 releases was very high. Thus, in reality, neither of these releases should have gone into production.

In terms of predicting the defect counts, our model successfully estimates defect ranges with 70 percent accuracy in the requirements and testing phases, whereas it estimates all defects with 100 percent accuracy in the development phase. Furthermore, when we observe confidence-level predictions of our model, it is seen that all releases have been assigned very low (VL) and low (L) confidence levels according to the company's predefined thresholds. These results lead to the following: If software managers used this kind of a model previously, all releases should have been improved or delayed until a desired confidence level was obtained. Managers can also monitor critical processes in the development life cycle so that corrective actions would be taken immediately. Finally, such predictive models can provide a funda-

mental input for the question of "when to release the software" as well as affect cost and opportunity considerations in software organizations.

Lessons Learned

There are certain challenges during the development and implementation of predictive models. During the development process, we easily collected software metrics using our open-source metric extraction tool, Prest. However, matching each defect with its corresponding file in order to form a training set for the model was a challenging task. To do this, companies have to store certain data in their systems. First, it is necessary to keep track of any bug/defect recorded during the testing phase through a bug-tracking system. Second, the changes applied on the source code due to a defect should be kept in a version history. Then, we can mine the version history to match every defect with all the files changed to fix the corresponding defect. After forming the training set, it is easy to apply any algorithm, and not necessarily only naïve Bayes, on the training set to learn the parameters. Software metrics required to form the testing set can be quickly collected with Prest.

During the implementation process, we must ensure at the beginning that the model yields the optimal prediction accuracy for the local data collected from the organization. Then, it is important to decide how and when a defect predictor would be used within the development life cycle. We suggest that such predictors should be used prior to the testing phase in order to guide the testers through defect-prone modules in the software system. We have integrated our model into the company's configuration management database (CMDB) system, which displays certain properties about the source code such as the difference between two releases, the complexity of the latest change, and added/deleted LOC during a given a time period (that is, release). Using our prediction model, this system also presents the probability of defect-proneness of any software module selected from the system. Thus, developers, as well as testers, can track the defect-proneness of their code in every release.

The application has been in use at the company for 12 months now. The company plans to improve the predictions by adding new metrics from version management systems.

From a practical point of view, this project led to new collaborations with global software organizations. Although AI techniques may be complex for practitioners, once the problems are identified carefully and tangible benefits are measured and monitored, the industry uses AI-based models very effectively.

Acknowledgements

This research is supported in part by Turkish State Planning Organization (DPT) under project number 2007K120610 and Turkcell Inc.

Notes

1. See D. Heeger, 1998, Signal Detection Theory, available at www.cns.nyu.edu/~david/handouts/sdt/sdt.html.

2. See G. Boetticher, T. Menzies, J. T. Ostrand, The PROMISE Repository of Empirical Software Engineering Data (promisedata.org/repository).

3. See the Software Engineering Institute (SEI), Team Software Process, Carnegie Mellon University (www.sei.cmu.edu/tsp).

References

Adrian, R. W.; Branstad, A. M.; and Cherniavsky, C. J. 1982. Validation, Verification, and Testing of Computer Software. *ACM Computing Surveys* (14)22: 159–192.

Alpaydin, E., ed. 2004. *Introduction to Machine Learning*. Cambridge, MA: The MIT Press.

Arisholm, E., and Briand, L.C. 2006. Predicting Fault-Prone Components in a Java Legacy System. In *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering,* 8–17. New York: Institute of Electrical and Electronics Engineers and Association for Computing Machinery.

Basili, V.; Briand, A.; and Melo, W. L. 1996. Validation of Object Oriented Design Metrics as Indicators of Quality Indicators. *IEEE Transactions on Software Engineering* (22)10: 751–761.

Brooks, A., ed. 1995. *The Mythical Man-Month: Essays on Software Engineering*. Reading, MA: Addison-Wesley.

Chidamber, S. R., and Kemerer, C. F. 1994. A Metrics Suite for OO Design. *IEEE Transactions on Software Engineering* (20)6: 476–493.

Fagan, M. 1976. Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems Journal* (15)3: 182–211.

Fenton, N., and Neil, M. 1999. A Critique of Software Defect Prediction Models. *IEEE Transactions on Software Engineering* (25)(5): 675–689.

Halstead, H. M., ed. 1977. *Elements of Software Science*. Amsterdam: Elsevier.

Heckerman, D. 1995. A Tutorial on Learning with Bayesian Networks. Technical Report. Redmond, WA: Microsoft Research.

Kocaguneli, E.; Tosun, A.; Bener, A.; Turhan, B.; and Caglayan, B. 2009. Prest: An Intelligent Software Metrics Extraction, Analysis, and Defect Prediction Tool. In *Proceedings of the 21st International Conference on Software Engineering and Knowledge Engineering,* 526–529. New York: Institute of Electrical and Electronics Engineers and Association for Computing Machinery.

Kuncheva, L. I. 2006. On the Optimality of Naïve Bayes with Dependent Binary Features. *Pattern Recognition Letters* (27)7: 830–837.

Lessmann, S.; Baesens, B.; Mues, C.; and Pietsch, S. 2008. Benchmarking Classification Models for Software Defect

Seventh AAAI Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE-11)

Save the Date! October 11–14, 2011

Please join us for AIIDE-11, to be held October 11-14, 2011 at Stanford University in Stanford, California. AIIDE is the definitive point of interaction between entertainment software developers interested in AI and academic and industrial AI researchers. While traditionally emphasizing commercial computer and video games, AIIDE invites researchers and developers to share their insights and cutting-edge results on all topics at the intersection of all forms of entertainment and artificial intelligence, including serious games, entertainment robotics, art, and beyond. The program will include invited speakers, research and industry presentations, project demonstrations, interactive poster sessions, and product exhibits. Registration information and other program details will be available on the AIIDE-11 website at www.aiide.org/aiide11 later this summer. Please send inquiries to aiide11@aaai.org, to conference chair Vadim Bulitko (University of Alberta), or to program chair Mark Riedl (Georgia Institute of Technology).

Prediction: A Proposed Framework and Novel Findings. *IEEE Transactions on Software Engineering* (34)4: 1–12.

McCabe, T. 1976. A Complexity Measure. *IEEE Transactions on Software Engineering* (2)4: 308–320.

Menzies, T.; Greenwald, J.; and Frank, A. 2007. Data Mining Static Code Attributes to Learn Defect Predictors. *IEEE Transactions on Software Engineering* (33)1: 2–13.

Moser, R.; Pedrycz, W.; and Succi, G. 2008. A Comparative Analysis of the Efficiency of Change Metrics and Static Code Attributes for Defect Prediction. In *Proceedings of the 30th International Conference on Software Engineering*, 181–190. New York: Institute of Electrical and Electronics Engineers and Association for Computing Machinery.

Munson, J. C., and Khoshgoftaar, T. M. 1992. The Detection of Fault-Prone Programs. *IEEE Transactions on Software Engineering* (18)5: 423–433.

Nagappan, N.; Ball, T.; and Murphy, B. 2006. Using Historical In-Process and Product Metrics for Early Estimation of Software Failures. In *Proceedings of the IEEE International Symposium on Software Reliability Engineering*. Piscataway, NJ: Institute of Electrical and Electronics Engineers.

Nagappan, N.; Murphy, B.; and Basili, V. 2008. The Influence of Organizational Structure on Software Quality: An Empirical Case Study. In *Proceedings of 30th International Conference on Software Engineering,* 521–530. New York: Institute of Electrical and Electronics Engineers and Association for Computing Machinery.

Ostrand, T. J.; Weyuker E. J.; and Bell, R. M. 2005. Predicting the Location and Number of Faults in Large Software Systems. *IEEE Transactions on Software Engineering* (31)4: 340–355.

Shull, F.; Boehm, V. B.; Brown, A.; Costa, P.; Lindvall, M.; Port, D.; Rus, I.; Tesoriero, R.; and Zelkowitz, M. 2002. What We Have Learned About Fighting Defects. In *Proceedings of the Eighth IEEE International Software Metrics Symposium,* 249–258. Piscataway, NJ: Institute of Electrical and Electronics Engineers.

Tosun, A.; Bener, A.; and Turhan, B. 2009. Practical Considerations in Deploying AI for Defect Prediction: A Case Study within the Telecommunication Industry. In *Proceedings of the 1st International Conference on Predictor Models (PROMISE)*. New York: Association for Computing Machinery.

Uyar, A.; Bener, A.; Ciray, H. N.; Bahceci, M. 2009. ROC Based Evaluation and Comparison of Classifiers for IVF Implantation Prediction. In *Proceedings of Second International ICST Conference on Electronic Healthcare for the 21st Century*, Lecture Notes in ICST. Berlin: Springer.

Zimmermann, T.; Weisgerber, P.; Diehl, S.; Zeller, A. 2004. Mining Version Histories to Guide Software Changes. In *Proceedings of the 26th International Conference on Software Engineering,* 563–572. Los Alamitos, CA: IEEE Computer Society.

Ayse Tosun Misirli is a research assistant and a PhD student in the Department of Computer Engineering at Bogazici University. She received her MS degree from the same department in 2008. She graduated from the School of Computer Science and Engineering at Sabanci University, Istanbul, Turkey, in 2006. Her research interests are software defect prediction, effort estimation, Bayesian modeling, and statistics. She is a student member of IEEE Computer Society and ACM SIGSOFT.

Ayse Bener is an associate professor in the Ted Rogers School of Information Technology Management at Ryerson University. Prior to joining Ryerson, Bener was a faculty member and vice chair in the Department of Computer Engineering at Bogazici University. Her research interests are software defect prediction, process improvement, software quality, and software economics. Bener has a PhD in information systems from the London School of Economics. She is a member of the IEEE, IEEE Computer Society, and the ACM.

Resat Kale is charging and collection unit manager at Turkcell Technology. Before taking this position, he had been working as a quality assurance manager at Turkcell Technology. He has a bacheler's degree in industrial engineering from Istanbul Technical University. He has been working in IT since 1992 in numerous positions. He has experience in software quality and testing. He is a member of the Turkish Testing Board (member of ISTQB).