

Estimating Fixing Effort and Schedule based on Defect Injection Distribution



Research Section

Qing Wang^{1*}, Lang Gou^{1,2}, Nan Jiang^{1,2}, Meiru Che^{1,2}, Ronghui Zhang^{1,2}, Yun Yang³ and Mingshu Li¹

¹ Institute of Software, Chinese Academy of Sciences, Beijing, China

² Graduate University of Chinese Academy of Sciences, Beijing, China

³ CITR, Swinburne University of Technology, Melbourne, Australia

Detecting and fixing defects are key activities in a testing process, which consume two kinds of skill sets. Unfortunately, many current leading software estimation methods, such as COCOMO II, mainly estimate the effort depending on the size of software, and allocate testing effort proportionally among various activities. Both efforts on detecting and fixing defects, are simply counted into software testing process/phase and cannot be estimated and managed satisfactorily. In fact, the activities for detecting defects and fixing them are quite different and need differently skilled people. The inadequate effort estimation leads to the difficulty of test process management. It is also the main problem which causes software project delays. In this article, we propose a method on Quantitatively Managing Testing (TestQM) process including identifying performance objectives, establishing a performance baseline, establish a process-performance model for fixing effort, and establishing a process-performance model for fixing the schedule, which supports high-level process management mentioned in Capability Maturity Model Integration (CMMI). In our method, defect injection distribution (DID) is used to derive estimation of fixing effort and schedule. The TestQM method has been successfully applied to a software organization for their quantitative management of testing process and proved to be helpful in estimating and controlling defects, effort and schedule of the testing process. Copyright © 2008 John Wiley & Sons, Ltd.

KEY WORDS: software measurement; quantitative process management; testing process; process-performance baseline; process-performance model

1. INTRODUCTION

Quantitative management is among the advanced features of highly mature processes as defined in

CMMI (Chrissis *et al.* 2006), which provides insight to the degree of goal fulfillment and root causes of significant process/product deviation.

Testing is an important method for quality control. There are substantial research on testing techniques and testing management method (Maximilien and Williams 2003, Gould *et al.* 2004, Clarke and Rosenblum 2006, Bertolino 2007). Testing is also an important process that needs to be managed quantitatively for high-maturity organizations. However,

* Correspondence to: Qing Wang, Laboratory for Internet Software Technologies, Institute of Software, The Chinese Academy of Sciences, No. 4 South Fourth Street, Zhong Guan Cun, Beijing 100080, China

† E-mail: wq@itechs.iscas.ac.cn



quantitative management method of the testing process is complex because it is constrained not only by the size of product, but also by the quality of prior activities in the lifecycle, such as design and coding. The more defects injected in the earlier activities, the more effort is needed to fix and verify them. How to estimate the effort and the defects-related data, establish a process-performance baseline (P-BL) and a process-performance model of testing process are the challenges. In fact, many software projects are delayed due to the slippage of the testing activities.

Many estimation methods estimate or predict the effort and defect separately. For example, COCOMO II (Boehm *et al.* 2000) is a famous cost estimation method with a family of extension models with respect to different types of development needs. CONstructive QUALity MODEL (COQUALMO) (Boehm *et al.* 2000) is one of these models which can be used to estimate the quality of software products in terms of defect density. But it does not consider the interrelationship between defect and effort. The testing effort comes mainly from the general percentage of the total estimated effort, which does not benefit high-level process management in CMMI.

As we know, even though there are many verification activities during the software development cycle, testing is still the most common and important method to detect and fix defects. The defects detected in testing are injected not only from coding, but also from requirements analysis and software design. In this article, we propose a method on Quantitatively Managing Testing (TestQM) process. Based on the TestQM method, the performance objectives of testing process are identified. Then some statistical techniques are used to analyze the data related to these objectives and some process-performance model are established. The TestQM method has been successfully applied to a software organization and appears to be very useful in helping software organizations quantitatively manage testing process.

The Institute of Software, Chinese Academy of Sciences (ISCAS) is a research and development organization in China. ISCAS developed a toolkit called SoftPM (Wang and Li 2005) which is used to manage software projects and has been deployed to many software organizations in China. The data used in this article comes from 17 projects based on facilitating SoftPM.

This article is organized as follows. The TestQM method is presented in Section 2. Section 3 introduces the empirical study on establishing quantitative management model for testing process based on the TestQM method. Section 4 describes the practice of quantitatively managing the testing process by using the model established in Section 3. Related work is discussed in Section 5. Section 6 summarizes our conclusions and points out future work.

2. THE METHODOLOGY

This section presents the TestQM method of quantitatively managing the testing process, which supports high-level process management mentioned in CMMI. As shown in Figure 1, the four steps of the TestQM method are to: (i) identify the performance objectives (P-Objs) to be managed quantitatively and construct data samples; (ii) establish the P-BL for the identified P-Objs; (iii) establish the process-performance model for fixing effort; and (iv) establish the process-performance model for fixing schedule.

As shown in Figure 1, by using the methodology, the empirically based models for the testing process can be established based on the analysis of historical data, which will be described in Section 3. Software projects can use the model to estimate and control the defects, effort and schedule quantitatively, which will be described in Section 4.

2.1. Identify P-Objs and Construct Data Samples

Normally, the effort of detecting and fixing defects, and the defect-injected phase are sensitive data that we should consider for testing process. A general assumption is that the effort of detecting and fixing defects should consume a certain percentage in the total development effort, and the effort of fixing defects is influenced by the defect number and the defect-injected phase. In the TestQM method, four P-Objs have been identified as follows:

1. Percentage of Detecting Effort (%Eff_{Detect}): Detecting effort means the effort for all detecting activities including test planning, test case preparation, test implementation and fix verification. The percentage of the detecting effort in the total effort is %Eff_{Detect}.
2. Defect Injection Distribution (DID): In general, many software organizations collect defect

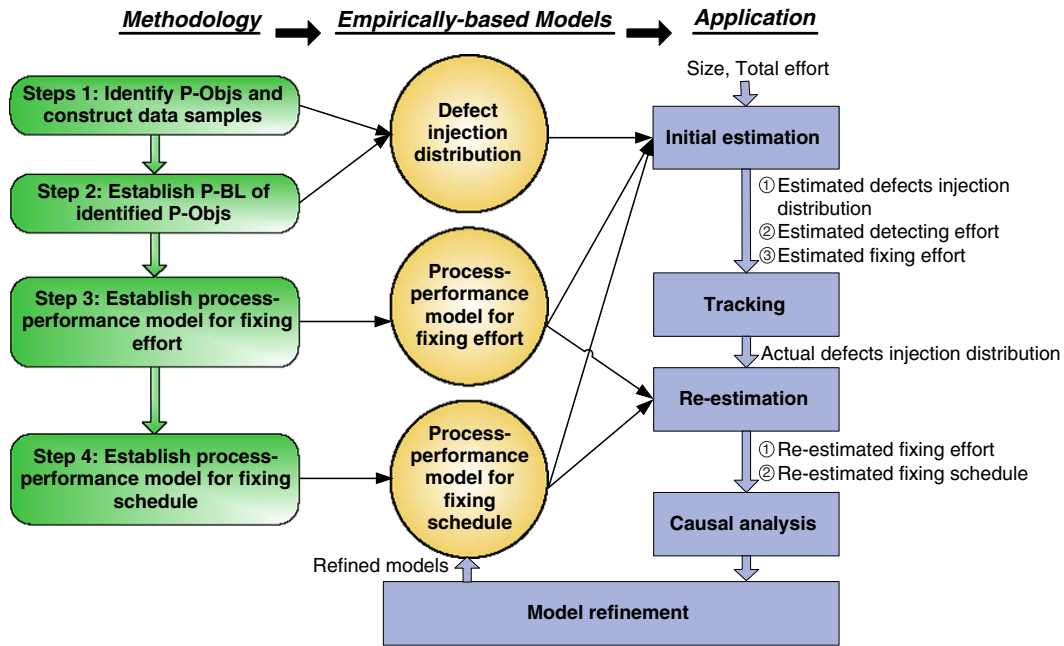


Figure 1. The illustration of the TestQM method

data for quality control. There are always some defects injected in the early phases, which are only detected during the testing activities, even in high-maturity organizations. In our method, three primary phases, namely requirements, design and coding, are used to classify the corresponding injected phases for each defect. The corresponding percentages of defects injected in these phases are denoted as: requirements, (%DI_{Req}); design, (%DI_{Design}); and coding, (%DI_{Code}) respectively.

The principles of assigning the injected phase are described as: (i) defect injected in the requirements phase: a defect that is due to poor requirements, such as inconsistent and unclear requirements;(ii) a defect injected in the design phase: a defect that is due to poor design, such as unclear interface, misunderstanding of requirements and incomplete data verification; and (iii) a defect injected in the coding phase: a defect that is due to poor coding, such as incorrect words in a Web page and inconsistent code against requirements or design.

3. Schedule Factor for Defect Fixing (SF_{Fix}). For each defect, the opening date is the day the defect is being submitted, and the closing date is the day the defect is being confirmed as

repaired. The schedule of defect fixing (Sced_{Fix}) can be calculated by the formula below.

$$Sced_{Fix} = \text{closing date} - \text{opening date} + 1.$$

Sometimes, certain defects are assigned 'deferred' and not to be fixed in the current release due to business pressures. In this case, we take the day the defect is being deferred and calculate the Sced_{Fix} as shown below.

$$Sced_{Fix} = \text{deferred date} - \text{opening date} + 1.$$

The Sced_{Fix} for deferred defects means the schedule of the defect being dealt with.

For each project, the average schedule of fixing one defect (ASced_{Fix}) injected in each phase can be calculated by the formula below.

$$ASced_{Fix} \text{ of each phase} = \frac{\text{total } Sced_{Fix}}{\text{total defects injected in the phase}}$$

Normally, the ASced_{Fix} of coding phase (ASced_{Code}) is the shortest. We use the ASced_{Code} as the benchmark (i.e. SF_{Code}), and calculate the ratio of ASced_{Fix} of requirements phase (ASced_{Req}) to ASced_{Code}, as well as the ratio of ASced_{Fix} of



design phase ($ASced_{Design}$) to $ASced_{Code}$ by the formula below.

$$SF_{Code} = 1$$

$$SF_{Req} = ASced_{Req} / ASced_{Code}$$

$$SF_{Design} = ASced_{Design} / ASced_{Code}$$

4. Percentage of Fixing Effort ($\%Eff_{Fix}$): Fixing effort data means the effort for all defect-fixing activities including defect analysis and fixing. $\%Eff_{Fix}$ is the percentage of the fixing effort in the total effort.

2.2. Establish P-BL of Identified P-Obj

P-BL is the basis for quantitative process management. As defined in CMMI, P-BL is a documented characterization of the actual results achieved by following a process, which is used as a benchmark for comparing actual process performance against expected process performance (Chrissis *et al.* 2006). P-BL is established based on the statistical analysis of historical data. There are many methods and techniques, such as Baseline-Statistic-Refinement (BSR) (Wang *et al.* 2006) and Statistical Process Control (SPC) (Florac and Careton 1999, Jalote and Saxena 2002) which can be used to establish P-BL.

Defect fixing is an important activity of software development, which demands a certain amount of effort. In the International Software Benchmark Standard Group (ISBSG), (www.isbsg.org), the fixing effort is collected and counted in rework effort. However, many effort estimation methods do not pay sufficient attention to the effort of defect fixing; instead, they just include it in the testing activities. Normally, defect detecting is performed by a testing team, and defect fixing is performed by a development team. Estimating their effort separately is helpful for an organization to plan its human resources and schedules. In addition, the fixing effort is strongly correlated with the number and injected phase of defects. Splitting them and establishing their P-BLs are very useful to manage testing process quantitatively.

For high-maturity software organizations, the defect-related process performance, such as defect injection, defect removal, and defect density, also has some common and stable properties. Many methods discuss the defect removal ratio and defect density. These are very useful and easy to

understand. Here we focus on the defect injection and the correlation between the defects and effort needed to fix them.

2.3. Establishing A Process-Performance Model for Fixing Effort

In CMMI, the process-performance model is a description of the relationships among attributes of a process and its work products that are developed from historical process-performance data, and calibrated using collected process and product measures from the project, and are used to predict results to be achieved by following a process (Chrissis *et al.* 2006).

In the testing activities, there is a consensus that the earlier a defect is injected, the more effort is needed to fix it. In contrast, the later a defect is injected, the less effort is needed to fix it. So, defects injected in an earlier phase, such as the requirements phase, have the effect of increasing the defect-fixing effort, whereas, defects injected in a later phase, such as the coding phase, have the effect of decreasing the defect-fixing effort.

After constructing defect-related data samples, software organizations can discover some more precise correlation between defects and fixing effort. The process-performance model for fixing effort is based on this hypothesis. There are some statistical methods which can be used to analyze the correlation between DID and $\%Eff_{Fix}$, such as multiple regression analysis (Wooldridge 2002). After the correlation between DID and $\%Eff_{Fix}$ has been analyzed, the regression equation between DID and $\%Eff_{Fix}$ can be used to refine the estimation of fixing effort after testing. The outcome can provide a guideline to estimate the effort of defect fixing based on the defects and the distribution of injection phases. So, after testing, project managers could reestimate and replan their fixing effort effectively. The factors of regression equation could be refined and calibrated based on the historical data of software organizations. Thereafter, it can be better applied in these organizations.

2.4. Establishing A Process-Performance Model for Fixing Schedule

As we mentioned before, many software projects are delayed due to the slippage of the testing process. In fact, many testing processes are delayed due to the



schedule overrun of defect-fixing activity. To solve this problem, TestQM uses a more effective method on estimating schedule of defect-fixing activity.

An algorithm to help estimate the schedule of defect-fixing activity is established based on the analysis of SF_{Fix} and the effort of defect fixing. The algorithm applies the following principles:

- Shortest schedule. Based on the total effort of defect fixing, the defect-fixing schedule should be as short as possible.
- Concurrent defect fixing. Defects which require a long fixing schedule should be fixed concurrently if there are sufficient human resources available.

The basic ideas of the algorithm are: (i) the fixing schedule of defects injected in requirements should be allocated first since the $ASced_{Req}$ is the longest, which is the basis of the fixing schedules of defects injected in design and coding; (ii) if the number of defects injected in the design and the number of defects injected in the coding are similar, as well as if the SF_{Req} is longer than the sum of SF_{Design} and SF_{Code} , then the fixing schedule of defects injected in design and coding could be allocated serially. Especially, in the algorithm, we assume that if $1/2 <$ numbers of defects injected in design/numbers of defects injected in coding < 2 , it means that the numbers of defects injected in design and coding are similar; and (iii) in the other cases, the schedule of defects injected in design and the schedule of defects injected in coding should be allocated concurrently.

The definitions used in the algorithm and the description of the algorithm are shown as follows:

- Assume the defects detected in testing process is $\{d_i\}$, $i = 1 \dots n$, n is the total number of defects detected in the testing process. Especially, $\{d_r\}$, $\{d_d\}$, $\{d_c\}$ denote the defects injected in requirements, design and coding phases respectively.
- Assume the numbers of defects injected in requirements, design and coding phases are D_r , D_d , D_c respectively.
- Assume $\forall i \in [1, n]$, s_i is the fixing schedule for d_i .
- Assume e is the effort of one day for a every full-time staff

Algorithm 1: Allocate the $Sced_{Fix}$.

Input: effort of defect fixing (E)

$ASced_{Req}$, $ASced_{Design}$, $ASced_{Code}$

SF_{Req} , SF_{Design} , SF_{Code}

Output: $Sced_{Fix}$ ($S = \{<d_1, s_1 >, <d_2, s_2 > \dots <d_n, s_n >\}$)

Steps: $S = \Phi$

1. If $D_r > 0$

Then allocate s_i for each d_r as concurrently as possible.

$S = S + <d_1, s_1 > + <d_2, s_2 > + \dots + <d_n, s_n >$

$d_i \in d_r \ i = 1 \dots D_r$

Assume S_r is the schedule for fixing all d_r

2. Else If $1/2 < D_d/D_c < 2$ and $SF_{Req} \geq SF_{Design} + SF_{Code}$

Then $M = \text{int}(SF_{Req}/(SF_{Design} + SF_{Code}))$

Assign no more than $M \times (d_d + d_c)$ to be fixed serially.

$S = S + <d_1, s_1 > + <d_2, s_2 > + \dots + <d_n, s_n >$

$d_i \in d_d + d_c \ i = 1 \dots D_d + D_c$

3. Else allocate the schedule for $\{d_d\}$ and $\{d_c\}$ concurrently.

$N = \text{int}(S_r/ASced_{Design})$

Assign no more than $N \times d_d$ to be fixed serially.

$S = S + <d_1, s_1 > + <d_2, s_2 > + \dots + <d_n, s_n >$

$d_i \in \{d_d\} \ i = 1 \dots D_d$

$N' = \text{int}(S_r/ASced_{Code})$

Assign no more than $N' \times d_c$ to be fixed serially.

$S = S + <d_1, s_1 > + <d_2, s_2 > + \dots + <d_n, s_n >$

$d_i \in \{d_c\} \ i = 1 \dots D_c$

4. If $E > \sum_{i=1}^n (s_i \times e)$ Then go to Step 1.

The P-BL of SF_{Fix} and the Algorithm 1 compose the process-performance model for fixing schedules.

Up to now, the P-BLs of the identified P-Objs, the correlations between the P-Objs, and the algorithm of estimating defect-fixing schedules compose the quantitative management model for the testing process. In the model, the P-BLs of $\%Eff_{Detect}$, DID and $\%Eff_{Fix}$, and the correlations between DID and $\%Eff_{Fix}$ are used for quantitatively controlling defects and effort of the testing process, while the P-BL of SF_{Fix} and the algorithm is used for managing the schedule of defect-fixing activities.



Table 1. Brief information about the 16 Web-based system development projects

Projects	Number of staff	Schedule (Months)	Size (KLOC)	Application domain	Process performance
1	12	12	151.9	Application system integration	Successful projects with little schedule overruns.
2	15	7	173.1	E-government	
3	5	6	18.2	E-government	
4	14	7.5	311.2	Application system integration	
5	6	4	76.9	Website design and development	
6	3	3	45.9	Website design and development	
7	6	2	17.0	Information management	All performed requirements, design, coding and testing processes
8	14	9	280.4	E-government	
9	4	5.5	45.0	E-government	
10	12	7	55.5	Information management	
11	9	5	60.7	E-government	
12	4	2	19.6	Information management	
13	11	9	90.4	Application system integration	
14	12	4.5	250.5	Application system integration	
15	5	6	80.0	Information management	
16	4	7	30.0	E-government	

3. EMPIRICALLY BASED MODELS

This section presents some empirical results on establishing a quantitative management model for the testing process by using the TestQM method presented in Section 2. We collected testing process data from 16 Web-based system development projects, and decided upon the empirically based models for testing process. Web-based development techniques have been widely applied in China. These 16 projects came from two closely related software organization entities. The two entities have the self-governed process management system and were rated at CMMI maturity level 3 and moving towards CMMI maturity level 4 during the period of our data collection.

Table 1 summarizes the brief information about the 16 projects. We collected the DID, %Eff_{Detect}, %Eff_{Fix} and SF_{Fix} data from the 16 projects indicated earlier. These data were reported by engineers and were collected in SoftPM (Wang and Li 2005).

3.1. Defect Injection Distribution

For the 16 projects, all the defects considered were detected in the testing activities. These defects were classified into four categories: critical defects, serious defects, noncritical defects and cosmetic defects. In this article, we only describe the total defects collected without distinguishing them. Table 2 shows the defects injected in three primary phases and the DIDs of the 16 projects.

The XmR (individuals and moving range) control chart (Grant and Leavenworth 1996, Florac and Careton 1999) is applied to analyze the DID data. Assume that the sequence of data sample is X_i , the moving range (mR) is:

$$mR_i = |X_i - X_{i-1}| \quad i = 2 \dots n$$

According to the theory of statistics, we can get the upper control limit (UCL), central line (CL), and lower control limit (LCL) for mR-chart and X-chart as follows:

$$UCL_{mR} = 3.268\overline{mR}, \quad CL_{mR} = \overline{mR}, \quad LCL_{mR} = 0$$

$$UCL_x = \overline{X} + 2.660\overline{mR}, \quad CL_x = \overline{X},$$

$$LCL_x = \overline{X} - 2.660\overline{mR}$$

Tables 3–5 show the XmR chart control limits. Figures 2–4 show the XmR control charts for %DI_{Req}, %DI_{Design} and %DI_{Code} respectively.

For the three XmR charts in Figures 2–4, all data points are distributed between the UCL and the LCL in both mR-chart and X-chart. Hence, the %DI_{Req}, %DI_{Design} and %DI_{Code} were converged and the distribution of defect injection appears to be stable. Therefore, the 14.9, 28.0 and 57.1% can be accepted as the P-BLs of %DI_{Req}, %DI_{Design} and %DI_{Code}, respectively, to be used to estimate the distribution of defect injection.



Table 2. Defects injected in each phase and the DIDs of the 16 projects

Projects	Number of defects injected in requirements	%DI _{Req}	Number of defects injected in design	%DI _{Design}	Number of defects injected in coding	%DI _{Code}
1	19	10.1	41	21.8	128	68.1
2	118	15.6	153	20.3	483	64.1
3	33	17.8	61	33.0	91	49.2
4	251	18.7	412	30.6	682	50.7
5	27	20.5	44	33.3	61	46.2
6	17	13.3	35	27.3	76	59.4
7	15	15.6	28	29.2	53	55.2
8	135	14.1	322	33.6	501	52.3
9	32	12.2	82	31.2	149	56.7
10	15	13.9	29	26.9	64	59.3
11	92	18.3	116	23.1	295	58.6
12	12	14.0	26	30.2	48	55.8
13	18	11.8	36	23.5	99	64.7
14	53	11.0	142	29.5	286	59.5
15	78	17.6	114	25.7	251	56.7
16	20	13.9	42	29.2	82	56.9

Table 3. XmR chart control limits for %DI_{Req} data

UCL _{mR}	CL _{mR}	LCL _{mR}	UCL _x	CL _x	LCL _x
10.2%	3.1%	0	23.2%	14.9%	6.6%

Table 4. XmR chart control limits for %DI_{Design} data

UCL _{mR}	CL _{mR}	LCL _{mR}	UCL _x	CL _x	LCL _x
15.1%	4.6%	0	40.3%	28.0%	15.8%

Table 5. XmR chart control limits for %DI_{Code} data

UCL _{mR}	CL _{mR}	LCL _{mR}	UCL _x	CL _x	LCL _x
15.9%	4.9%	0	70.0%	57.1%	44.2%

3.2. Process-Performance Model for Fixing Effort

We collected the total effort (in labor hour), defect-detecting effort (in labor hour) and defect-fixing effort (in labor hour) of the 16 projects. Unfortunately, the detecting effort and fixing effort for individual defects were not recorded. Hence we could only collect the total detecting effort and total fixing effort for the 16 projects. Then, we calculated %Eff_{Detect} and %Eff_{Fix} of the 16 projects. Table 6 shows the total effort, detecting effort, fixing effort, %Eff_{Detect} and %Eff_{Fix} of the 16 projects.

First, we analyze the %Eff_{Detect} data. The XmR chart control limits for %Eff_{Detect} data are shown in

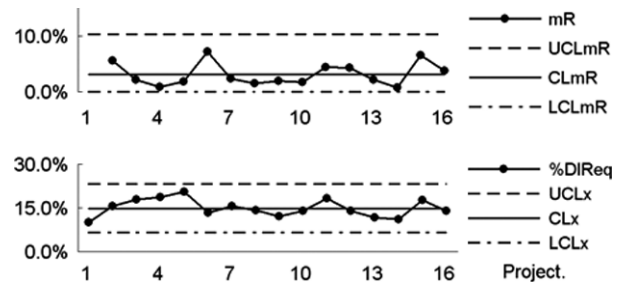


Figure 2. XmR chart for %DI_{Req} data of the 16 projects

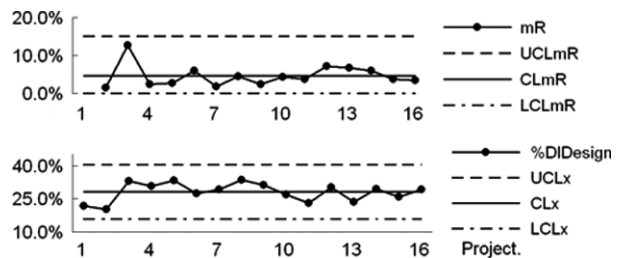


Figure 3. XmR chart for %DI_{Design} data of the 16 projects

Table 7. We construct the XmR chart in Figure 5 by using the %Eff_{Detect} data in Table 6 and control limits in Table 7. For both the mR-chart and X-chart, all data points distributed between the UCL and the LCL. The process appears to be stable. The CL_x (22.9%) can be considered as the P-BL of %Eff_{Detect} to be used to estimate defect-detecting effort.

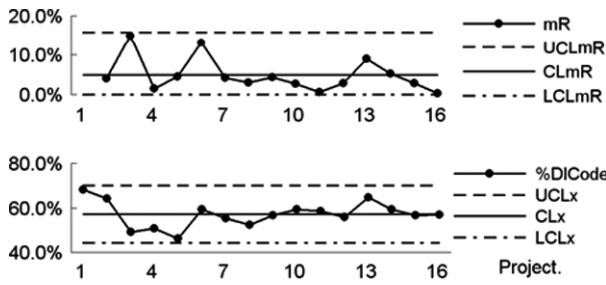


Figure 4. XmR chart for %DI_{Code} data of the 16 projects

Table 6. Total effort, detecting effort, fixing effort, %Eff_{Detect} and %Eff_{Fix} of the 16 projects

Projects	Total effort	Detecting effort	Fixing effort	%Eff _{Detect}	%Eff _{Fix}
1	7,048	1,396	762	19.8	10.8
2	11,614	3,734	2,370	32.2	20.4
3	3,143	785	632	25.0	20.1
4	11,177	3,624	2,839	32.4	25.4
5	2,926	609	536	20.8	18.3
6	1,313	182	108	13.9	8.2
7	1,560	354	245	22.7	15.7
8	10,865	2,566	1,521	23.6	14.0
9	3,397	693	420	20.4	12.4
10	7,114	1,070	1,403	15.0	19.7
11	6,864	1,684	1,325	24.5	19.3
12	1,205	265	185	22.0	15.4
13	14,683	2,684	2,214	18.3	15.1
14	6,579	2,117	824	32.2	12.5
15	4,230	940	802	22.2	19.0
16	1,934	401	264	20.7	13.7

Table 7. XmR chart control limits for %Eff_{Detect} data

UCL _{mR}	CL _{mR}	LCL _{mR}	UCL _x	CL _x	LCL _x
22.9%	7.0%	0	41.5%	22.9%	4.2%

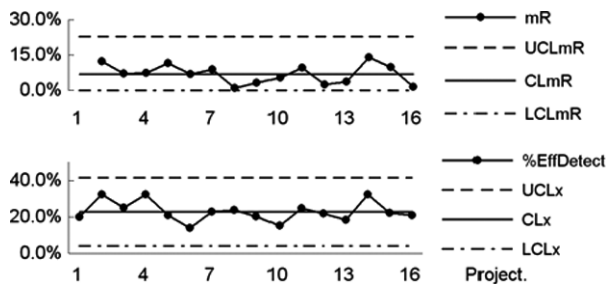


Figure 5. XmR chart for %Eff_{Detect} data of the 16 projects

Next, we analyze the %Eff_{Fix} data. Here also, we use the XmR control chart. Table 8 shows

the XmR chart control limits and Figure 6 shows the XmR control chart for %Eff_{Fix}. As shown in Figure 6, the %Eff_{Fix} also appears to be stable and converged. In this case, the CL_x (16.2%) can be treated as the P-BL of %Eff_{Fix} to be used to estimate the effort of defect fixing of testing process.

Finally, we establish the process-performance model for fixing effort. As mentioned in Section 2.3, DID will influence the %Eff_{Fix}. In this section, we analyze the correlation between %Eff_{Fix} and DID. Figures 7–9 are the scatter diagrams of %DI_{Req}, %DI_{Design}, %DI_{Code} and %Eff_{Fix} data based on Tables 2 and 6. In Figure 7, as expected, %Eff_{Fix} increased with %DI_{Req}, which means that %DI_{Req} and %Eff_{Fix} have positive correlation; and in Figure 9, %Eff_{Fix} decreased with %DI_{Code}, which means that %DI_{Code} and %Eff_{Fix} have negative correlation. In Figure 8, there is no obvious relationship between %DI_{Design} and %Eff_{Fix}, which means that %DI_{Design} and %Eff_{Fix} are uncorrelated.

In detail, we analyze the multiple correlations between %DI_{Req}, %DI_{Code} and %Eff_{Fix} by using

Table 8. XmR chart control limits for %Eff_{Fix} data

UCL _{mR}	CL _{mR}	LCL _{mR}	UCL _x	CL _x	LCL _x
15.1%	4.6%	0	28.6%	16.2%	3.9%

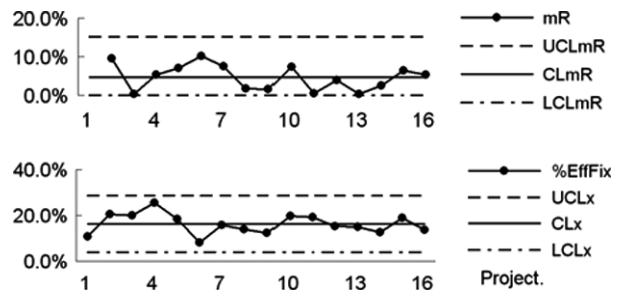


Figure 6. XmR chart for %Eff_{Fix} data of the 16 projects

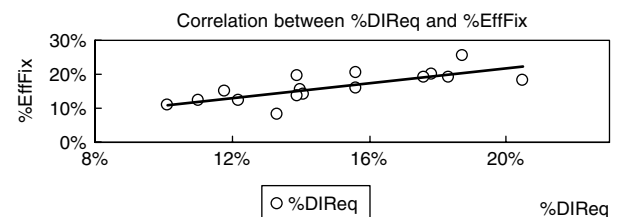


Figure 7. Correlation between %DI_{Req} and %Eff_{Fix}

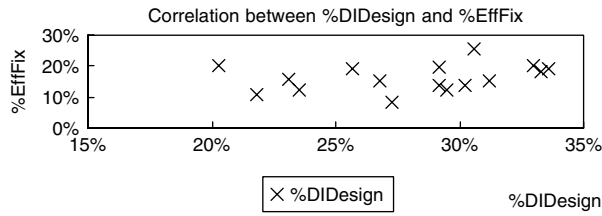


Figure 8. Correlation between %DI_{Design} and %Eff_{Fix}

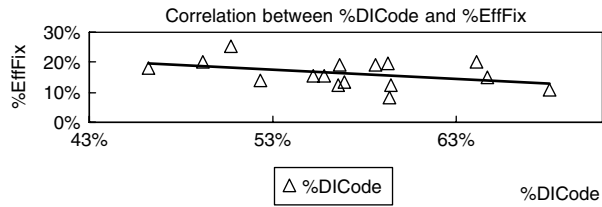


Figure 9. Correlation between %DI_{Code} and %Eff_{Fix}

multiple linear regression. Let X_R, X_C, Y denote the dataset on %DI_{Req}, %DI_{Code} and %Eff_{Fix} of the 16 projects based on Tables 2 and 6 respectively. By performing linear regression on independent variables X_R, X_C and dependent variable Y using Matlab 6.1 (<http://www.mathworks.com>), we first derive the binary linear regression equation as follows:

$$Y = -0.1597 + 1.3712 \times X_R + 0.2065 \times X_C$$

Table 9. Total Sced_{Fix}, ASced_{Fix} and SF_{Fix} of each phase in the 16 projects

Projects	Requirements			Design			Coding		
	Total Sced _{Fix}	ASced _{Fix}	SF _{Req}	Total Sced _{Fix}	ASced _{Fix}	SF _{Design}	Total Sced _{Fix}	ASced _{Fix}	SF _{Code}
1	455	23.95	4.20	607	14.80	2.60	729	5.70	1
2	3351	28.40	3.43	3261	21.31	2.57	4001	8.28	1
3	762	23.09	2.79	764	6.95	1.32	754	8.29	1
4	6071	24.19	4.76	2862	6.95	1.37	3465	5.08	1
5	272	10.07	2.94	208	4.73	1.38	209	3.43	1
6	297	17.47	3.95	166	4.74	1.07	336	4.42	1
7	92	6.13	3.78	87	3.11	1.91	86	1.62	1
8	2975	22.04	3.85	2596	8.06	1.41	2867	5.72	1
9	750	23.44	4.33	712	8.68	1.61	806	5.41	1
10	101	6.73	3.85	115	3.97	2.27	112	1.75	1
11	1358	14.76	4.86	706	6.09	2.00	896	3.04	1
12	206	17.17	2.99	294	11.31	1.97	276	5.75	1
13	601	33.39	3.87	578	16.06	1.86	854	8.63	1
14	675	12.74	3.69	609	4.29	1.24	987	3.45	1
15	1231	15.78	3.97	743	6.52	1.64	997	3.97	1
16	436	21.80	3.27	553	13.17	1.98	546	6.66	1

Then, an F-test (Wooldridge 2002) is performed. As calculated by Matlab 6.1, we have $F = 9.5484$. Let n denote the number of data points which is equal to 16, and k denote the number of independent variables which is equal to 2. At the confidence level $\alpha = 0.05$, the critical value of $F_{\alpha=0.05}(k, n-k-1) = F_{\alpha=0.05}(2, 13) = 3.81$. It is clear that $F_{\alpha=0.05}(2, 13) < F$. Therefore, the correlation between %DI_{Req}, %DI_{Design} and %Eff_{Fix} is linearly prominent. The regression equation between %DI_{Req}, %DI_{Code} and %Eff_{Fix} can be used to adjust the estimation of defect-fixing effort after testing.

3.3. Process-Performance Model for Fixing Schedule

We collected Sced_{Fix} for each defect and calculated the total Sced_{Fix} of defects injected in each phase. Then, we calculated the ASced_{Fix} of each phase. We used the ASced_{Fix} of the coding phase (named SF_{Code}) as the benchmark, and calculated the SF_{Req} and SF_{Design} following equations discussed in Section 2.1. Table 9 shows the total Sced_{Fix}, ASced_{Fix} and SF_{Fix} of each phase in the 16 projects.

We use the XmR control chart to analyze the distribution of SF_{Fix}. Tables 10 and 11 show the XmR chart control limits. Figures 10 and 11 show the XmR control charts for SF_{Req} and SF_{Design} respectively. For the two XmR charts, all data points are distributed between the UCL and the LCL in both mR-chart and X-chart. Hence, the SF_{Req} and



SF_{Design} were converged and the distribution of the schedule factor of defect fixing appears to be stable. Therefore, 3.78, 1.77 and 1 can be accepted as the P-BLs of SF_{Req}, SF_{Design} and SF_{Code} respectively to be used to estimate the Sced_{Fix} for the testing process.

According to P-BLs of SF_{Fix}, it is obvious that the earlier in the phase the defects get injected, the longer is the schedule needed to fix the defects. Based on the ASced_{Fix} of the 16 projects and the P-BL of SF_{Fix}, the organization defined some rules for defects management as shown in Table 12.

Based on the P-BLs of SF_{Fix}, we establish the process-performance model for fixing the schedule. According to Algorithm 1 in Section 2.4, some parameters in the Algorithm can be specified using

Table 10. XmR chart control limits for SF_{Req} data

UCL _{mR}	CL _{mR}	LCL _{mR}	UCL _x	CL _x	LCL _x
2.69	0.82	0	5.98	3.78	1.59

Table 11. XmR chart control limits for SF_{Design} data

UCL _{mR}	CL _{mR}	LCL _{mR}	UCL _x	CL _x	LCL _x
1.20	0.37	0	2.75	1.77	0.80

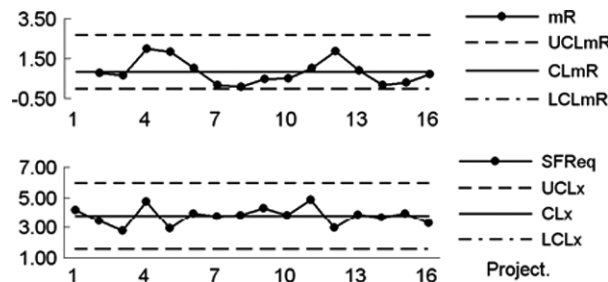


Figure 10. XmR chart for SF_{Req} data of the 16 projects

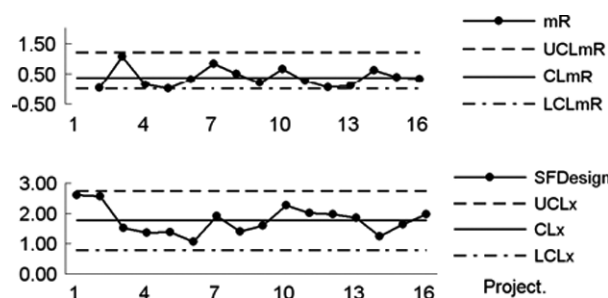


Figure 11. XmR chart for SF_{Design} data of the 16 projects

Table 12. Defects management rules

Rules	Description
Early detection principle	Defects should be detected as early as possible.
Limitation of defect-fixing schedule	Defects injected in the requirement phase should be fixed in 8 days; the control limits are 5–11 days.
	Defects injected in the design phase should be fixed in 5 days; the control limits are 3–7 days.
	Defects injected in the coding phase should be fixed in 2 days; the control limits are 1–3 days.

Table 13. Initial P-BLs of the P-Objs

P-Objs	Initial P-BLs
%DI _{Req} , %DI _{Design} , %DI _{Code}	14.9, 28.0, 57.1
%Eff _{Detect} , %Eff _{Fix}	22.9, 16.2
SF _{Req} , SF _{Design} , SF _{Code}	3.78, 1.77, 1

the P-BLs of SF_{Fix} and the defects management rules as shown below.

$$ASced_{Req} = 8, ASced_{Design} = 5, ASced_{Code} = 2$$

$$SF_{Req} = 3.78, SF_{Design} = 1.77, SF_{Code} = 1$$

Later projects can apply the process-performance model for fixing schedule by using the above parameters.

Up to now, the initial P-BLs of all identified P-Objs could be established as shown in Table 13.

Based on the empirical study described in Sections 3.1–3.3, the organization established the empirically based models for the testing process, which includes the experience results of DID, the process-performance models for fixing effort and schedule. Later projects can use the models to quantitatively manage the defects, effort and schedule for testing process.

4. APPLICATION OF TESTQM

The organization applied the empirically based models established in Section 3 on an ongoing project. The steps of applying the empirically based models for the testing process are: (i) based on the P-BLs of the P-Objs, estimating the defect-detecting effort, defect-fixing effort and number of defects



Table 14. Extended P-BLs of Web-based system development projects

Detected defect density - DDD	Software productivity - Prod	%DI _{Req} , %DI _{Design} , %DI _{Code}	%Eff _{Detect} , %Eff _{Fix}	SF _{Req} , SF _{Design} , SF _{Code}
4.01 Defects/KLOC	2.3 KLOC/Labor Month	14.9, 28.0, 57.1	22.9, 16.2	3.78, 1.77, 1

injected in each phase during the project planning; (ii) through the testing activities, collecting the defect-related data and re-estimating the schedule and effort of defect fixing when the actual P-Objs has abnormality; and (iii) after the testing process, refining the models if the testing process is normal.

4.1. Initial Estimation from P-BLs

As mentioned earlier, the organization was rated at CMMI maturity level 3 and was moving to CMMI maturity level 4. It had some P-BLs in place, such as detected defect density and software productivity. The detected defect density refers to (defects/code size) where the defects are detected in testing activities (except unit testing). The indicator of detected defect density is used to control the quality of software before being submitted for testing. The software productivity is the mean productivity (total size/total effort) which can be used to estimate the total effort. Besides these, we added the process-performance model presented in Section 3 to optimize project management. The new extended P-BLs of Web-based system development projects in the organization are shown in Table 14 with some new quantitatively controlled objectives defined.

An ongoing Web-based system development project, named TM, was selected in the organization. Table 15 summarizes the brief information about the project.

Table 15. Brief information about project TM

No. of staff	Plan schedule	Plan size	Development approach
8	4 months	67 KLOC	Increment and Iteration

Project TM was planned to complete the whole software product through two iterations. Each iteration implemented half the product functions. Before the first iteration started, the project manager and skilled engineers estimated the sizes of both iterations. Then, the total defects detected in the testing activities were estimated by using formula: Size × Detected defect density (DDD); the total effort was estimated by using formula: Size/Prod. After that, the estimation for both iterations could be elaborated further. Especially, Project TM applied the defects management rules (Table 12), which was established based on the P-BLs of SF_{Fix}, and used the rules to manage the schedule of defect-fixing activity. Table 16 shows the estimation results for the two iterations of project TM.

Based on the estimation, the project manager of project TM established a project plan for both iterations and performed the exercise.

4.2. Tracking

During the testing activities, the defects-related data were collected in SoftPM. The P-Objs were

Table 16. Initial Estimation for each iteration of project TM

Estimation	1st iteration	2nd iteration
Size (KLOC)	30	37
Schedule (Months)	2	2.4
Total defects detected in testing activities (Size × PRDE)	120	148
Defects injected in requirements (Total defects × %DI _{Req})	18	22
Defects injected in design (Total defects × %DI _{Design})	34	42
Defects injected in coding (Total defects × %DI _{Code})	68	84
Total effort (labor month) (Size/Prod)	13.0	16.1
Detecting effort (labor month) (Total effort × %Eff _{Detect})	3.0	3.7
Fixing effort (labor month) (Total effort × %Eff _{Fix})	2.1	2.6
Development effort (labor month) (Total effort × (1 - %Eff _{Detect} - %Eff _{Fix}))	7.9	9.8



Table 17. Actual performance data of the project TM

Actual performance data	1st iteration	2nd iteration
Size (KLOC)	28	34
Schedule (months)	2.5	1.9
Total defects detected in testing activities	138	132
Defects injected in requirements	46	18
Defects injected in design	30	40
Defects injected in coding	62	74
Total effort (labor month)	20	14
Detecting effort (labor month)	4.4	3
Fixing effort (labor month)	7.1	2.5
Development effort (labor month)	8.5	8.5

calculated correspondingly. Table 17 shows the actual performance of the two iterations in project TM.

4.3. Fixing Effort Re-estimation

During the testing activities of the first iteration, the defects injected in the requirements, design and coding phases were 46, 30 and 62 respectively as shown in Table 17. Correspondingly, %DI_{Req} (X_R), %DI_{Design} and %DI_{Code} (X_C) were 33.3, 21.8 and 44.9%, respectively. Compared to the P-BLs in Table 14 and the control limits in Table 3, %DI_{Req} was higher, which means more defects were injected in the requirements phase. Given this abnormality, the project manager performed some further analysis. As mentioned earlier, the defect-fixing effort should be greater due to the larger number of defects injected in the requirements phase. The %Eff_{Fix} (Y) was reestimated based on the regression equation ($Y = -0.1597 + 1.3712 \times X_R + 0.2065 \times X_C$). The new %Eff_{Fix} was 38.7%. The reestimated fixing effort was extended from 2.1 labor months to 6.9 labor months.

4.4. Fixing Schedule Re-estimation

With the fixing effort increased, the schedule of the defect-fixing activity in the first iteration should be extended. Based on the process-performance model for fixing the schedule, the schedule of fixing defects injected in requirements, design and coding phase was allocated as follows:

First, the schedule of fixing defects injected in requirements was allocated. For the 46 defects injected in requirements, there are 4 defects which must be fixed serially since they are correlated, the other defects can be fixed concurrently. Based

on this situation, the 46 defects were each divided into 12 groups, where the first 11 groups included 4 defects and the last group included 2 defects. Defects in the same group were assigned to be fixed serially, whereas, defects in different groups were fixed concurrently. Since the ASced_{Req} was 8 days, the S_r was 32 days.

Then, the schedule of fixing defects injected in the design and coding phases was allocated. Since $D_c = 62/D_d = 30 > 2$, the schedule of fixing defects injected in the design and coding phases was allocated separately. For defects injected in the design phase, $N = \text{int}(S_r/ASced_{\text{Design}}) = \text{int}(32/5) = 6$, which means, no more than 6 defects could be fixed serially. For defects injected in the coding phase, $N' = \text{int}(S_r/ASced_{\text{Code}}) = \text{int}(32/2) = 16$, which means no more than 16 defects could be fixed serially. According to the above analysis, the Sced_{Fix} was reestimated as shown in Figure 12.

In Figure 12, the schedule was described by a Gantt view in SoftPM (Wang and Li 2005), based on Algorithm 1, $E < \sum_{i=1}^n (s_i \times e)$, so the schedule shown in Figure 12 is applicable. Based on the reestimated defect-fixing schedule, the schedule of the first iteration had to be delayed by 10 working days. In addition, one engineer was added to fix the defects.

4.5. Causal Analysis

Since the actual %DI_{Req} of the first iteration was higher, the processes of requirements development, requirements management, especially the requirements review may have some problems. In reality, there could be many possible causes leading to the poor quality of the requirements phase. We analyzed all 46 defects injected in the requirements phase, and used a Pareto diagram to rate the major causes, as shown in Figure 13. In Figure 13, almost 80% of the 46 defects were due to the first two causes: unclear requirements and inconsistent requirements. Based on the causal analysis, the organization improved the requirements review process.

During the second iteration, the defects-related data were collected (Table 17). The defects injected in the requirements, design and coding phases were similar to the estimation. Therefore, we did not need to reestimate the fixing effort and schedule. The second iteration was completed on time, and the actual performance (Table 17) was similar to the

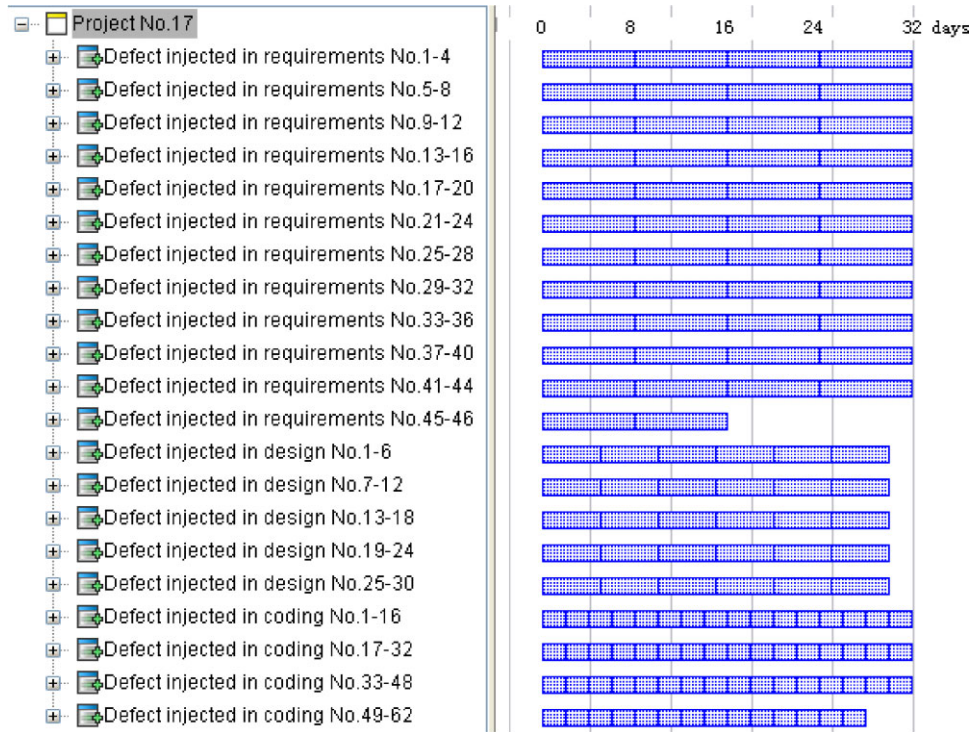


Figure 12. Reestimated defect-fixing schedule

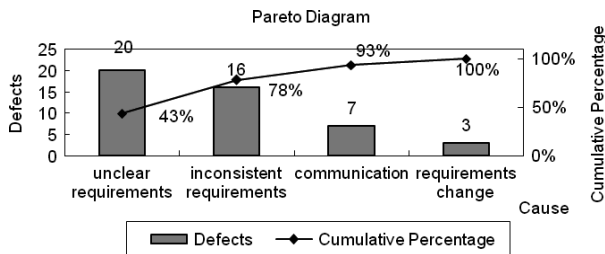


Figure 13. Causal analysis of poor requirements

estimation (Table 16). Hence, the testing process of the second iteration is normal and stable.

4.6. Model Refinement

After project TM was finished, the organization added its data to the historical data and refined the empirically based models established in Section 3 by using the data of the second iteration in project TM, since the testing process in the second iteration of project TM was normal and stable.

Table 18 shows the actual performance data of the second iteration in project TM. Combined with

the performance data of the 16 projects (Tables 2, 6, and 9), we can refine the P-BLs of the P-Objs.

Similar to Section 3, we applied XmR control charts to analyze the distribution of $\%Eff_{Detect}$, DID, SF_{Fix} and $\%Eff_{Fix}$ data in the 17 projects. Due to space limitation, we do not present the XmR control charts. As a result, in each XmR control chart, all data points are distributed between the UCL and the LCL in both mR-chart and X-chart. Hence, the $\%Eff_{Detect}$, DID, SF_{Fix} and $\%Eff_{Fix}$ data were converged and stable. Therefore, we can refine the P-BLs of $\%Eff_{Detect}$, DID, SF_{Fix} and $\%Eff_{Fix}$ as shown in Table 19.

Since the refined P-BLs were established based on more data than the initial P-BLs (Table 13), the

Table 18. Actual performance data of the second iteration in project TM

P-Objs	Actual performance
$\%Eff_{Detect}$	21.4
$\%DI_{Req}$, $\%DI_{Design}$, $\%DI_{Code}$	13.6, 30.3, 56.1
SF_{Req} , SF_{Design} , SF_{Code}	3.24, 1.80, 1
$\%Eff_{Fix}$	17.9



Table 19. Refined P-BLs of the P-Objs

P-Objs	Refined P-BLs
%DI _{Req} , %DI _{Design} , %DI _{Code}	14.8, 28.2, 57.0
%Eff _{Detect} , %Eff _{Fix}	22.8, 16.3
SF _{Req} , SF _{Design} , SF _{Code}	3.75, 1.78, 1

refined P-BLs can quantitatively manage testing process more precisely. In the future, when the organization has more project data of testing process, they can refine the P-BLs for %Eff_{Detect}, DID, SF_{Fix} and %Eff_{Fix} continuously. Based on the refined P-BL of SF_{Fix}, the parameters of the process-performance model for fixing the schedule are refined correspondingly.

We use the actual performance data of the second iteration in project TM (Table 18) to refine the correlation regression between %DI_{Req}, %DI_{Design} and %Eff_{Fix} as we described in Section 3.2.

We analyze the multiple correlations between %DI_{Req}, %DI_{Code} and %Eff_{Fix} by using multiple linear regression. Let X_R , X_C , Y denote the dataset on %DI_{Req}, %DI_{Code} and %Eff_{Fix} of the 17 projects respectively. By performing linear regression on the independent variables X_R , X_C and the dependent variable Y , we first derive the refined binary linear regression equation as follows:

$$Y = -0.1249 + 1.2910 \times X_R + 0.1700 \times X_C$$

Then, an F -test (Wooldridge 2002) is performed. As calculated by Matlab 6.1, we get the F statistic $F = 8.8700$. Let n denote the number of data points which is equal to 17, and k denote the number of independent variables which is equal to 2. At the confidence level $\alpha = 0.05$, the critical value of $F_{\alpha=0.05}(k, n-k-1) = F_{\alpha=0.05}(2, 14) = 3.74$. It is clear that $F_{\alpha=0.05}(2, 14) < F$. Therefore, the refined correlation between %DI_{Req}, %DI_{Design} and %Eff_{Fix} is linearly prominent.

Up to now, the models established in Section 3 have been refined. The experience from this case study validates that the TestQM method presented in Section 2 and the models established in Section 3 are helpful for improving quantitatively managing testing process. And the TestQM method can be used in initial estimation, tracking the process performance, identifying abnormality of process, analyzing the causes, reestimating the fixing effort and schedule, and improving the process to keep it controllable.

5. RELATED WORK

CONstructive COSt MOdel (COCOMO) II (Boehm *et al.* 2000) is a widely used estimation model, which allows one to estimate the total effort of a project depending on the estimated size. It provides two sets of empirical results on effort distribution for both waterfall and RUP lifecycle phases, which can be used to estimate effort of each phase including testing activities proportionally. COCOMO II cannot predict the effort of defect detecting and fixing accurately. COQUALMO (Boehm *et al.* 2000) is a quality-model extension to COCOMO II. It is used to estimate defects injected in different activities, and defects removed by defect removal activities. COQUALMO does not associate the defects with the effort of defect fixing.

Software Productivity Research (SPR) (Jones 2000) (<http://www.spr.com>) is a provider of consulting services to help companies manage software development processes. SPR collected data from about 9000 projects and reported the percentages of the testing effort for system software, military software, commercial software, MIS and outsourcing software. Mizuno *et al.* (2002) develop a linear multiple regression model of estimating the testing effort. In the model, the testing effort can be obtained from the design and review efforts, and are also influenced by historical data factors. Both of them do not distinguish the effort of detecting from the effort of fixing in the testing activities.

The Rayleigh model (Norden 1963, Putnam 1987, Kan 2002) is based on Weibull's statistical distribution. Supported by a large body of empirical data, it is found that the defect detecting or removal patterns follow Rayleigh's curve. In this way, the Rayleigh model can be used for predicting the potential software defects (Putnam and Meyers 1991). It can be concluded from the Rayleigh model that there are some defects injected in early phases left to later phases such as the testing activities.

The related work mentioned above shows that the defects-related data have been paid much attention by both academia and industry. In addition, there is much research on defect distribution and testing effort. Unfortunately, the above methods do not distinguish the effort of defect detecting from the effort of defect fixing. They also do not present mechanisms to adjust the effort of defect fixing based on the defect distribution. In this article, we focus on identifying more performance



objectives to indicate the relationship between the effort, schedule and defects, which is valuable for quantitative testing process management.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we propose a method, named TestQM for quantitatively managing testing process. The method includes identifying performance objectives (P-Objs), establishing a P-BL, establishing a process-performance model for fixing effort, and establishing a process-performance model for fixing the schedule.

From the empirical study, we find that the DID of the requirements, design and coding phases have common and stable properties for high-maturity software organizations, and the schedule factor of defect fixing (SF_{Fix}) in different projects appears to be stable. In addition, the percentages of the detecting effort ($\%Eff_{Detect}$) and fixing effort ($\%Eff_{Fix}$) are also similar. With the analysis of multiple regression, some correlations emerge between the effort of defect fixing and the DID. Based on the analysis of the SF_{Fix} , an algorithm was established for estimating the schedule of defect-fixing activity.

Based on the method, a software organization established empirically based models for testing process, quantitatively controlled an ongoing project, and refined the models. Through the application, we can conclude that the TestQM method is effective in quantitatively managing testing process. The TestQM method also provides helpful insights for project managers to make the detailed estimation for testing process, such as the distribution of defect injection, the effort for detecting and fixing defects, and the schedule of defect-fixing activity.

As future work, the TestQM method addressed in the article can be refined with more studies and practices in different application domains. Some other factors should be considered. For example, the differences in project size, personnel capability and project type may affect the P-BLs of $\%Eff_{Detect}$, DID, SF_{Fix} and $\%Eff_{Fix}$.

ACKNOWLEDGEMENTS

This article is based on an earlier version (Wang *et al.* 2007). It was supported partly by the National

Natural Science Foundation of China (Grant Number: 60573082 and 60473060), National Hi-tech Research and Development Program of China (Grant Number: 2007AA010303), and National Basic Research Program of China (Grant Number: 2007CB310802). One of the authors, Yun Yang, gratefully acknowledges the support of the K. C. Wong Education Foundation, Hong Kong. We are grateful for Ye Yang's help in improving the article, as well as anonymous reviewers' comments on the early version of this article (Wang *et al.* 2007).

REFERENCES

- Bertolino A. 2007. Software testing research: achievements, challenges, dreams. *Proceedings of 29th International Conference on Software Engineering, Future of Software Engineering*, Minneapolis; 85–103.
- Boehm BW, Horowitz E, Madachy R, Reifer D, Clark BK, Steece B, Brown AW, Chulani S, Abts C. 2000. *Software Cost Estimation with COCOMO II*. Prentice Hall PTR: Upper Saddle River, NJ.
- Chrissis MB, Konrad M, Shrum S. 2006. *CMMI(R): Guidelines for Process Integration and Product Improvement*. Addison-Wesley Publishing Company: Boston, MA.
- Clarke LA, Rosenblum DS. 2006. A historical perspective on runtime assertion checking in software development. *SIGSOFT Software Engineering Notes* 31(3): 25–37.
- Florac A, Careton WD. 1999. *Measuring Software Process-Statistical Process Control for Software Process Improvement*. Addison-Wesley Professional: Reading, MA.
- Gould C, Su ZD, Devanbu P. 2004. Static checking of dynamically generated queries in database applications. *Proceedings of the 26th International Conference on Software Engineer*, Edinburgh, 645–654.
- Grant E, Leavenworth R. 1996. *Statistical Quality Control*, 7th edn. McGraw-Hill: New York.
- Jalote P, Saxena A. 2002. Optimum control limits for employing statistical process control in software process. *IEEE Transactions on Software Engineering* 28: 1126–1134.
- Jones C. 2000. *Software Assessments, Benchmarks, and Best Practices*. Addison-Wesley Professional: Boston, MA.
- Kan SH. 2002. *Metrics and Models in Software Quality Engineering*. Addison-Wesley Professional: Reading, MA.
- Maximilien EM, Williams L. 2003. Assessing Test-Driven Development at IBM. *Proceedings of the 25th International Conference on Software Engineering*, Portland, 564–569.



Mizuno O, Shigematsu E, Takagi Y, Kikuno T. 2002. On estimating testing effort needed to assure field quality in software development. *Proceedings of the 13th International Symposium on Software Reliability Engineering*, Annapolis, 139–146.

Norden PV. 1963. *Useful Tools for Project Management, Operations Research in Research and Development*. John Wiley and Sons: New York.

Putnam LH. 1987. A general empirical solution to the macro software sizing and estimating problem. *IEEE Transactions on Software Engineering* **SE-4**: 345–361.

Putnam LH, Meyers W. 1991. *Measures for Excellence: Reliable Software on Time, Within Budget*. Prentice Hall PTR: Englewood Cliffs, NJ.

Wang Q, Li M. 2005. Measuring and improving software process in China. *Proceedings of the 4th International*

Symposium on Empirical Software Engineering, Australia, 183–192.

Wang Q, Jiang N, Gou L, Liu X, Li M, Wang Y. 2006. BSR: a statistic-based approach for establishing and refining software process performance baseline. *Proceedings of the 28th International Conference on Software Engineering*, Shanghai, 585–594.

Wang Q, Gou L, Jiang N, Che M, Zhang R, Yang Y, Li M. 2007. An empirical study on establishing quantitative management model for testing process. *Proceedings of the International Conference on Software Process*, Minneapolis, 233–245.

Wooldridge J. 2002. *Introductory Econometrics: A Modern Approach*. South-Western College Pub.