OPTIMAL REFACTORING POLICY FOR AGILE INFORMATION SYSTEMS MAINTENANCE: A CONTROL THEORETIC APPROACH

Research-in-Progress

Jimmy SJ. Ren
Department of Information Systems,
City University of Hong Kong
83 Tat Chee Avenue, Hong Kong
sjren2@student.cityu.edu.hk

Wei Wang
Department of Information Systems,
City University of Hong Kong
83 Tat Chee Avenue, Hong Kong
wewang8@student.cityu.edu.hk

Zhimin Hua
Department of Information Systems,
City University of Hong Kong
83 Tat Chee Avenue, Hong Kong
zmhua2@student.cityu.edu.hk

Kaiquan Xu
Department of Information Systems,
City University of Hong Kong
83 Tat Chee Avenue, Hong Kong
xkaiquan2@student.cityu.edu.hk

Stephen SY Liao
Department of Information Systems,
City University of Hong Kong
83 Tat Chee Avenue, Hong Kong
issliao@cityu.edu.hk

Abstract

Many information systems development companies are facing the question on how to apply agile methods in information systems maintenance (ISM). Performing correction of software defects in ISM inevitably degenerates program structure. On the other hand, agile methods provide refactoring to improve program structure without changing its behavior. This paper builds an optimal control model to balance the trade-off between defect correction and refactoring. We answer three questions. First, is that optimal to perform parallel defect correction and refactoring? Second, how to determine the iteration length for agile ISM if team wants to include refactoring in the iteration? Third, how long the iteration should be if team wants to improve program's structure to a certain level at the end of the iteration? To our knowledge, this paper is the pioneer in understanding agile ISM policy analytically. Managerial implications of the results are also discussed in the paper.

Keywords: Agile maintenance, optimal control, refactoring, defect correction

Introduction

In the last decade, a set of information systems development (ISD) methods, collectively labeled agile methods, were proposed and prevalent in the ISD community. These methods include eXtreme Programming (XP), Scrum, Crystal, Agile modeling, Feature Driven Design, etc. (Dybå and Dingsøyr 2008). The main motivation of using agile methods originates from the unsatisfactory performance of the traditional ISD methods, the "waterfall" model as a representative, in the current increasingly turbulent business environment. In such a business environment, it is not uncommon to encounter issues like quickly evolving requirements, constantly changing stakeholder preferences, as well as time-to-market pressures. Values and practices in traditional ISD methods such as thorough plan, strict control and non-iterative manner are challenged by these issues mainly because of the lack of flexibility and ability to adapt to change under uncertainty (Conboy 2009). It is worth mentioning that though different agile methods are available, they share common characteristics in facilitating flexibility (Maruping et al. 2009).

As agile methods were gradually adopted by many ISD companies in the industry, such methods are not only used in the development stage of the software life cycle but other stages such as information systems maintenance (ISM) as well. This is simply because business agility in general is not only required in the development stage but every stage of the software life cycle (Auer and Miller 2001). It's widely accepted that software maintenance is one of the most important stages in the software life cycle. The main reason is the maintenance stage usually occupies a large portion of time in the whole cycle. Besides, the cost of the maintenance stage also contributes much to the whole expense of a project. Studies reported that in many software projects, the maintenance cost constitutes more than half of the total expense of the projects (Banker et al. 1993, Lientz and Swanson 1981). With the continuous prevalence of agile methods, it is believed that more ISM practitioners will tend to use agile as the maintenance process because using two distinct processes within an organization generally lowers the business efficiency (Senge 2006, Svensson & Host 2005). Therefore, there is a need to re-scrutinize maintenance in the agile context.

However, despite the importance of investigating how agile methods interact with maintenance, not many studies in the extant literature can be found to show how to manage agile methods under ISM context. In this paper, we study two important competing efforts, namely refactoring and defect correction, in agile maintenance by an analytical model. The analysis shall answer fundamental questions of using agile methods in ISM. While the theoretical analysis should fill the current research gap in agile maintenance, the managerial implications generated by the study should be of interest to ISM practitioners as well.

The rest of the paper arranges as the following. In the second section we review the literature, identify the research gap and form the research questions. In the third section we present an optimal control model and its solution process to solve the research questions. In the fourth section, we highlighted our contributions and summarize the managerial implications. We also discuss the limitations, the unfinished work and how we are going to complete the work in the last section.

Literature Review

Empirical evidences can be found in the literature to justify the usefulness of agile methods in ISM. Svensson & Host (2005) conducted a case study in a software maintenance and evolution organization and results showed that a large number of agile practices such as planning game, pair programming and continuous integration are appropriate in maintenance context and facilitate the transition of knowledge. Shaw (2007) studied the application of agile practices in a maintenance project and also found many agile practices were effective in maintenance work. In addition, the study showed that to do the maintenance in short iterations conforms to the nature of ISM. Rico (2008) also looked at agile methods in the maintenance context and argued that ISM practitioners would benefit from many agile practices including refactoring. The study also reported a case in which the code size in the project was reduced over 40 percent by using refactoring. Though these empirical studies may confirm the effectiveness of agile methods in maintenance context, however, no general rule could be found to guide the operation of agile maintenance. Therefore, while confidence of using agile methods maybe gained from these studies, practitioners can hardly find any concrete rules to follow. On the other hand, due to the lack of theoretical implications in current studies, it is not easy to generalize the research result to a wider context or to form the theoretical foundation for other researchers. Given the status of the extant literature on agile

maintenance, together with the importance of investigating this topic which we discussed in the introduction section, we believe the research gap in understanding how to use agile methods in ISM effectively is significant.

In order to fill the gap and conduct a theoretical analysis, we first turn to maintenance literature to seek evidence of important factors in maintenance. We found that correcting defects and improving software attributes are two of the most important tasks in software maintenance (Kemerer and Slaughter 1997). Usually, the software sent to the maintenance team is far from "bug free", therefore it is maintenance team's job to further reduce the defects in the software. We can also conclude from some more recent studies that similar to the regular maintenance projects, software defect correction occupies much of the workload in agile maintenance as well (Poole et al. 2001, Shaw 2007, Rico 2008, Svensson & Host 2005). The other major maintenance task, improving software attributes is equally important. The key attribute to address in the maintenance stage is maintainability which in the source code level, means the quality of software's internal structure (Kitchenham et al. 1999, Oman and Hagemeister 1992). In agile practices, refactoring is the one to improve software structure (Fowler et al. 1999).

Refactoring is a well-established technique to improve software maintainability by improving program structure and fixing implicit problems without changing software's behavior (Griswold 1991). In the maintenance project, maintainers usually focus on correcting specific defects and overlook the overall structure of the program. Studies showed that defect correction will inevitably degenerate the system architecture, creates more implicit issues and eventually makes the system hard to maintain (Griswold 1991, Opdyke 1992). Maruping et al. (2009) investigated typical agile practices including refactoring in the eXtreme Programming context within a large scale company. They argued that refactoring shall improve the program structure and reduce various aspects of software complexity. Their results showed that the overall software complexity is indeed reduced after adopting agile methods. While the overall defects in the system are reduced, there is no evidence to show refactoring explicitly addresses these defects. Similar argument could be concluded in the study in another large company under the scenario of ISD (Fitzgerald et al. 2006). Within the development procedure, programmers restructured the system, without modifying functionalities, to improve non-functional aspects such as duplication of code, simplicity and flexibility. By continuously doing this during the development the potential implicit structural problems were successfully reduced.

If take a more careful scrutiny, it is not hard to discern the trade-off between refactoring and defect correction. If refactoring is not performed, system architecture tends to be degenerated rapidly by defect correction in agile projects (Auer and Miller 2001). Worse system structure directly lowers maintenance team's agility since implicit structural issues make defects become harder to correct and code becomes harder to understand. Then slower reaction to the new defect reports, which are imposed by the customers, could be expected. In agile, refactoring is used to increase agility by fixing implicit structural issues, namely improving the design. Fowler et al. (1999) defined the rule on when to conduct refactoring in general. They proposed that at the time you add functions, need to fix a bug and do code review, refactoring should be carried out. It is important to see that all of these three activities previously mentioned could be found in defect correction activities in agile maintenance projects (Poole et al. 2001). This reaffirms the necessity of conducting refactoring if defect correction occurs in the maintenance. Fowler et al. (1999) also pointed out that refactoring is a regular practice which is expected to be carried out often. However, there's one cost that refactoring has to pay, that is refactoring doesn't directly address explicit defects in the system. If refactoring is improperly conducted the efficiency and effectiveness of the overall maintenance may be negatively affected. For example, if too much effort is devoted to refactoring, the progress of defect correction may be delayed.

In order to get a better understanding of such a trade-off in agile maintenance, we visualize it by drawing the following diagram.

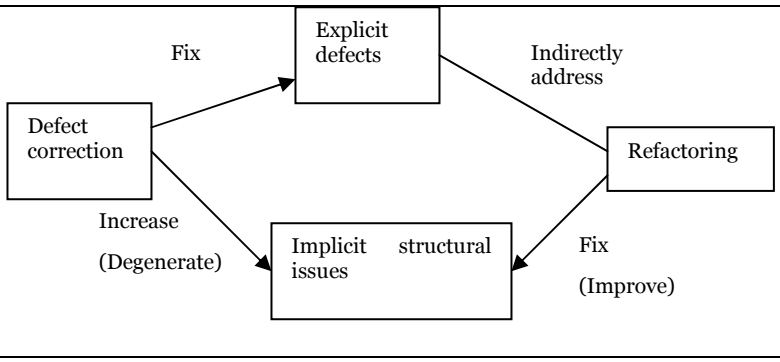

Figure 1. Trade-off between defect correction and refactoring in agile maintenance

So we can conclude that refactoring is not only critical to software maintenance in practice, it also has the nature of competing with defect correction. Therefore, there is a call for investigation on the detailed relationship between refactoring and defect correction.

In the previous discussion, we identified a research gap in understanding how to use agile methods in the maintenance context. Particularly, we found the trade-off between refactoring and defect correction is a key problem to tackle in order to fill this gap and therefore this problem is of our interest. However, there is little research in the extant literature to study how to balance such an important trade-off to minimize the program structural issues. Because optimal control theory is an effective tool to investigate trade-offs between variables (Chiang 1992), this paper attempts to use an optimal control model to answer three basic questions in balancing this trade-off. First, is that optimal to perform parallel defect correction and refactoring? Second, how to determine the iteration length for Agile maintenance project if the project team wants to include refactoring activities in the iteration? Third, how long the iteration should be if team wants to improve program's structure to a certain level at the end of the iteration?

Optimal Control Model and the Solution Process

Optimal control model

Ji et al. (2005) used an optimal control model to solve the trade-off between software construction and bug fixing. Though this model was not designed for agile maintenance projects, we found that this model conceptually fits the agile maintenance context. Therefore, we draw from the structure of the model and build an optimal control model to solve our problem. We first present the notations used in the model in Table 1.

Table 1. Notations used in the optimal control model

Notation	Definition	Remarks
b	Number of explicit defects to be fixed in the system	State variable
s	Number of structural problems in the system	State variable
u	Effort on defect correction	Control variable
$1-u$	Effort on refactoring correction	
T	Time horizon of one iteration	
B	Number of defects left in the system at the beginning of the iteration	

$\tilde{B}$	Number of defects left in the system at the end of the iteration	
a_0 to a_5	Constant coefficients	

The detailed setting of the model is presented as the following.

Minimize $S = s(T)$, **(1)**

Subject to $\dot{b} = -(\frac{a_0 bu}{a_1})$, **(2)**

$\dot{s} = (a_2 + a_3 b + a_4 s)(-\dot{b}) - (a_5 s)(1-u)$, **(3)**

and $b(0) = B$, $b(T) = \tilde{B}$, $b \geq 0$, $s(0) = 0$, $0 \leq u \leq 1$. **(4)**

S is a functional represents the number of program's structural problems. As we indicated previously, the objective functional (1) is to minimize the number of program's structural problems at the end of the iteration. We then explain the first state equation. Defects in the maintenance project normally decreases, thus the rate of change of defects should be negative. This explains the negative sign in (1). Maintenance team tends to fix the "easy but important" defects first and leave the "hard to fix and less important" defects to the end. Therefore, the more defects the system has in the system, the faster the defects are to be fixed. The constant a_0 is the defect complexity coefficient. It captures the extent to which the correction process of defects needs to involve different components in the system. The more components the correction is involved, the more complex the defect is to be fixed (less a_0). So, smaller a_0 leads to slower defect fixing speed. Control variable u is the effort to be devoted in defect correction. Larger u leads to higher rate of change of defects, this is reasonable because more defect correction speed is gained if more effort is devoted to defect fixing. In the denominator, constant a_1 is the initial design quality factor, higher value of a_1 represents more serious structural problems in the system at the beginning of the iteration. It is expected that more structural problems cause lower defect correction rate. From (4) we can show that this equation is a fixed endpoint problem.

We also interpret the rate of change of structural issues' number in the system as the rate of change of system degeneration. In the second equation, two factors are connected to this rate. First, the faster the maintenance team corrects the defects, the more likely the team is to degenerate the program structure. Second, if more effort is allocated to refactoring, the program structure tends to be better and the rate of change of the program degeneration will therefore be decreased. Such relationship explains the negative sign between the two portions in the equation. On the other hand, given the same defect correction speed, if the maintenance team's defect correction quality is high (lower a_2), program degeneration tends to be slower. Thus, a_2 is the defect correction quality coefficient. a_3 captures the extent to which the defects intertwine with each other. Therefore $a_3 b$ measures the degree of cohesion among defects, the higher the cohesion is the faster the system tends to be degenerated when fixing these defects. And a_4 is the system quality coefficient. The number of structural problems in the system is proportional to the extent to which the system is degenerated by code modification. The more serious the system is degenerated, the faster it will be degenerated again. On the other hand, given the same refactoring effort $1-u$, the less structured the program is, the more potential benefit the refactoring could bring. Therefore, the refactoring will be more effective if the program is more contaminated. Thus, a_5 is the refactoring effectiveness coefficient and $a_5 s$ measures the effectiveness of refactoring.

In the model, we assume that at the beginning of the maintenance project, the program does not have structural issues, therefore $s(0) = 0$. Finally, we need to impose the constraint to the control variable $0 \le u \le 1$.

The solution process

We formulate the Hamiltonian function as the following

$$H = \lambda_1 \dot{b} + \lambda_2 \dot{s} = \lambda_1 (-\frac{a_0 bu}{a_1}) + \lambda_2 [(a_2 + a_3 b + a_4 s)(-\dot{b}) - (a_5 s)(1-u)]. \quad \textbf{(5)}$$

The costate variables λ_1 and λ_2 can then be shown

$$\dot{\lambda}_1 = -\frac{\partial H}{\partial b} = \lambda_1 \frac{a_0 u}{a_1} - \lambda_2 (\frac{a_0 a_2 u + 2a_0 a_3 bu + a_0 a_4 su}{a_1}), \quad \textbf{(6)}$$

$$\dot{\lambda}_2 = -\frac{\partial H}{\partial s} = \lambda_2 (a_5 - \frac{a_0 a_4 bu}{a_1} - a_5 u). \quad \textbf{(7)}$$

To minimize the objective functional is to maximize the Hamiltonian function with respect to u. We can observe that H is linear in u, therefore the optimal solution for the control variable in general is

$$u^*(t) = \begin{cases} 1 & H_u > 0 \\ \textit{to be known} & H_u = 0 \\ 0 & H_u < 0 \end{cases}. \quad \textbf{(8)}$$

We can provide the economic interpretations of the costate variables λ_1, λ_2 and the Hamiltonian function. Because our objective functional is to minimize the program's structural problems with the constraint of finishing the defect correction work in the iteration. This objective reflects the value of agile maintenance activity. Therefore, the costate variable $\lambda_1(t)$ is the shadow value of software defects at time t. And the costate variable $\lambda_2(t)$ is the shadow value of program's structural problems at time t. We can now provide the economic interpretations of the Hamiltonian function as the sum of rate of change of defect value and structural problem value which represents the future value effect of the control variable u.

Because at the beginning of the iteration, the structure of the program is not degenerated by the defect correction activity, so all the effort should be devoted to the defect correction. It means the optimal solution for the control variable u should be set to 1 at the beginning of the maintenance. Thus, we plug in $u^* = 1$ to (2), the differential equation of the state variable b, we solve the equation and the solution is

$b(t) = Be^{-\frac{a_0}{a_1}t}$. **(9)** The solution shows the number of defects decreases exponentially with time t. Such path of defect correction is consistent with classical empirical study (Belady and Lehman 1976) and software reliability model (Goel and Okumoto 1979).

We then try to solve the equation (3), the differential equation for the state variable z and it can be shown in the following general form

$\dot{s}(t) = p(t)s(t) + q(t)$, $s(t_1) = s_1$. **(10)** We now solve this general differential equation. Let $u = e^{-\int p(t)dt}$, then multiplies both side of the equation and integrate both sides, we get

$\int (s(t)e^{-\int p(t)dt})'dt = \int q(t)e^{-\int p(t)dt}dt$. Then we can get from the previous equation that $s(t) = e^{\int_0^t p(x)dx}(\int_0^t q(x)e^{-\int_0^x p(y)dy}dx + C)$, C is a constant. Plug in the initial condition $s(t_1) = s_1$ we get $C = (s_1 - e^{\int_0^{t_1} p(x)dx}\int_0^{t_1} q(x)e^{-\int_0^x p(y)dy}dx)e^{-\int_0^{t_1} p(x)dx}$. Plug in C back, we get the solution as the following

$s(t) = e^{\int_0^t p(x)dx}(s_1 e^{-\int_0^{t_1} p(x)dx} + \int_{t_1}^t q(x)e^{-\int_0^x p(y)dy}dx)$. **(11)** If we plug in $u^* = 1$ to (3), the differential equation of s, and according to the general form (10), we can let $p(t) = -\dot{b}(t)a_4$, $q(t) = -\dot{b}(a_2 + a_3 b)$. Plug in them to the general solution form (11), we can get the specific solution of s

$$s(t) = (1 - e^{-a_4(b(t)-B)})(\frac{a_3 - a_2 a_4}{a_4^{\,2}}) - \frac{a_3}{a_4}(b(t) - Be^{-a_4(b(t)-B)}). \quad \textbf{(12)}$$

The expression (12) represents the path of system degeneration at the first stage of the maintenance. We can show in (12) that s is increasing with t. This solution is consistent with our speculation, because pure defect correction will generate more structural problems in the system.

When the maintenance team has corrected some defects, the system structure will be degenerated to a certain extend governed by the path described in (12). In order to finish the defect correction task and, at the same time, to improve the system structure, there will be three possible actions the maintenance team can choose in the next period of time. We assume the workload of defect correction planed in the iteration is reasonable. Figure 2 shows these three choices.

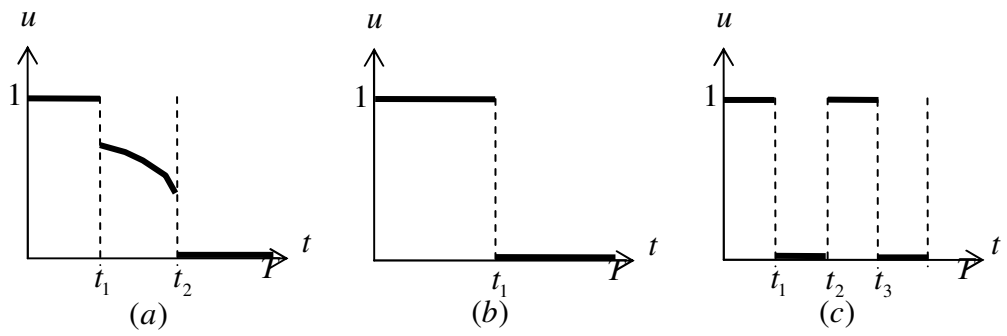

Figure 2. Three possible actions in the second stage of the iteration

The first possible action, showed in figure 2(a), is to perform defect correction and refactoring simultaneously (from time t_1 to t_2) and switch to pure refactoring once all the defects are corrected in t_2. The second possible action, showed in figure 2(b), is to finish the defect correction first and then switch to pure refactoring. The third one, showed in figure 2(c), is to switch to refactoring right away and switch back to defect correction when appropriate thereafter. Then to switch between defect correction and refactoring back and forth in the iteration until all the defects are corrected.

Next we need to find out which of the three possible actions is optimal. We start with the first possible action, to do the refactoring and defect correction simultaneously in the next stage. According to the general solution form for the control variable u, if we would like to allocate defect correction and refactoring simultaneously, the control variable needs to satisfy $0 < u < 1$. From (8), we can show this

means H_u has to be equal to 0. And if we would like to keep the value of H_u zero, the derivative of H_u with respect to time t, $\dot{H}_u$, has to be 0. We first get the expression for H_u

$H_u = -\lambda_1 \frac{a_0 b}{a_1} + \lambda_2[\frac{(a_2 + a_3 b + a_4 z)a_0 b}{a_1} + a_5 s]$. **(13)** If we let $\frac{a_0 b}{a_1} = V$, $\frac{(a_2 + a_3 b + a_4 s)a_0 b}{a_1} + a_5 s = R$, we can get $\dot{H}_u = -(\lambda_1 \dot{V} + \dot{\lambda}_1 V) + \lambda_2 \dot{R} + \dot{\lambda}_2 R$. **(14)** Then we plug in $\dot{\lambda}_1$, $\dot{\lambda}_2$, $\dot{b}$, $\dot{s}$, and through some careful calculation we are able to show that $\dot{H}_u = \lambda_2(\frac{a_0 a_2 a_5 b}{a_1} + \frac{a_0 a_3 a_5 b^2}{a_1})$. **(15)** If we let $\dot{H}_u = 0$, since $\lambda_2 \neq 0$, all the constants $a_i > 0$ and $b \geq 0$, the only possibility is $b = 0$. This means if to perform defect correction and refactoring simultaneously is optimal, the necessary condition is the number of defect needs to be 0. This obviously contradicts to the first possible action we just mentioned, because if the number of defect is 0, we don't need to perform defect correction anymore. However, this result still can't differentiate whether the second or the third possible action, is optimal because both the second and the third actions do not require performing defect correction and refactoring at the same time.

In order to judge whether the second or the third possible action is optimal, we need to determine the sign of λ_2. To see the sign of λ_2, we draw from the variational view of optimal control (Chiang 1992). Because the objective functional is to minimize $z(T)$, therefore the terminal condition of λ_2 is $\lambda_2(T) = -1$. And with the aid of this terminal condition we can solve the differential equation for λ_2, $\lambda_2(t) = -e^{\int (t-T)(a_5 - \frac{a_0 a_4 b u}{a_1} - a_5 u)dt}$. **(16)** Therefore, we can show that $\lambda_2 < 0$ throughout the maintenance period. We already know that at the beginning of the maintenance period, $u = 1$. That means $H_u > 0$. From (8) we also know that when the control variable switches from 1 to 0, the sign of H_u must switch from positive to negative. Since now we know $\lambda_2 < 0$ at all time, we can know that the rate of change of H_u witch respect to time t, $\dot{H}_u$ is always less than zero. Then it's not possible for H_u to be larger than zero again once its sign becomes negative from positive. Now we can eventually eliminate the third possible action.

If the team always wants refactoring activity to be able to occur in the iteration, it can generate the threshold time by using equation (9). If we plug in $b(t_1) = \tilde{B}$ to equation (9) we get $t_1 = -\frac{a_1}{a_0}\ln(\frac{\tilde{B}}{B})$. **(17)** t_1 is a positive value because $\tilde{B} < B$ and $\ln(\frac{\tilde{B}}{B}) < 0$. The length of the iteration must be longer than the time determined by t_1 to allow refactoring to be able to occur in the iteration.

In addition, if the team always wants to improve program's structure to a certain level $\tilde{s}$ and wants to determine the time t_2 for that. This could be done by solving equation (3) by plugging in $u = 1$. If we plug in $u = 1$, the solution of (3) is $s(t) = Ce^{-a_5 t}$, **(18)** with the initial condition $s(t_1) = (1 - e^{-a_4(b(t_1)-B)})(\frac{a_3 - a_2 a_4}{a_4^2}) - \frac{a_3}{a_4}(b(t_1) - Be^{-a_4(b(t_1)-B)})$. **(19)** Plug in (19) to (18) to solve C we get $C = s(t_1)e^{a_5 t_1}$. Then we can finally get the solution for (18) $s(t) = s(t_1)e^{a_5 t_1}e^{-a_5 t}$. **(20)** With (20), if we let $\tilde{s} = s(t_2)$, we can easily solve for t_2 and $t_1 + t_2$ is the duration length which enables the team to

achieve certain structural requirement imposed on the program structure s at the end of the iteration. So far, three research questions raised in the previous sections can all be answered.

Managerial Implications and Contributions

Several managerial implications can be generated from our research result. First of all, our finding suggests maintenance managers or agile coach of agile ISM projects should not plan tasks which conduct parallel defect correction and refactoring. Maintainers should also try not to perform parallel defect correction and refactoring in their daily work. Secondly, the length of the iteration is usually determined empirically. Team usually proposes tentative iteration durations and tries to fit the work into such duration. Adjustments may be made if team fails to achieve the iteration goal. There was no tool or analytical method to support the team in determining the iteration length in maintenance projects. Our study provides an analytical method to aid the determination of iteration length.

There are two major contributions in this study. Firstly, while many researchers focused on ISM under traditional process and some of the other studies tried to understand agile maintenance empirically, there is few theoretical analysis research studies maintenance in agile context. Our work is the pioneer study to gain theoretical insights of agile maintenance analytically. Our results not only address key issues of filling the research gap we identified, it can also form the theoretical foundation for other researchers in the future study. Secondly, our findings and proposed managerial implications may benefit the maintenance practitioners who want to adopt agile methods in work.

Limitations and Future Work

We studied the trade-off between defect correction and refactoring. We found it is not optimal to perform parallel defect correction and refactoring. This is an interesting finding because the conventional wisdom tends to believe that parallel work or multi-tasking should be superior in terms of the overall performance. However, our finding denies this intuition and it suggests that agile maintenance team should always try to correct all the defects listed in the correction plan of the current iteration and leave the rest of the time to refactoring.

The majority portion of the unfinished work is a numerical experiment. We didn't perform such an experiment to gain deeper insights on how every constant coefficient influences the variables such as the time threshold in the model. Numerical experiment shall enable us not only be able to understand the phenomenon analytically but also approach the problem quantitatively. More interesting findings could be generated from such numerical experiment. For example, we may answer the question "to what extent the defect complexity effects the overall program structure?".

There are several limitations in the current study. First, agile maintenance is an iterative process. In this study we assume the maintenance work in each iteration shares the same or at least a similar pattern. Such assumption may threat the validity of the findings when special requirements are imposed to some iteration. For instance, in some iteration, customers may require the team to fix much more defects than regular to meet some particular deadline. In the future work, we need to consider the situation in more iterations to see whether the result will change significantly.

Second, in equation (2) of the current model, only the constant a_1 is used to reflect how structural issues affect the defect correction rate. This may be considered insufficient since the program's structure is actually a dynamic variable rather than a constant. If the change in program's structure significantly affects the defect correction rate, the validity of the findings may be threatened. In the future study, we plan to integrate the second state variable s with equation (2). We expect the solution of such a modified model will disclose more insights of agile maintenance.

References

Auer, K. and R. Miller. 2001. *Extreme Programming Applied: Playing to Win*, Addison-Wesley Professional.

Banker, R., S. Datar, et al. 1993. "Software complexity and maintenance costs," *Communications of the ACM* (36:11), pp. 81-94

Belady, L. A. and M. M. Lehman. 1976. "A model of large program development," *IBM Systems Journal* (15:3), pp. 225-252.

Chiang, A. C. 1992. *Elements of Dynamic Optimization*, McGraw-Hill.

Conboy, K. 2009. "Agility from First Principles: Reconstructing the Concept of Agility in Information Systems Development," *Information Systems Research* (20:3), pp. 329-354.

Dybå, T. and T. Dingsøyr. 2008. "Empirical studies of agile software development: A systematic review," *Information and Software Technology* (50:9-10), pp. 833-859.

Fitzgerald, B., G. Hartnett, et al. 2006. "Customising agile methods to software practices at Intel Shannon," *European Journal of Information Systems* (15:2), pp. 200-213.

Fowler, M., K. Beck, et al. 1999. *Refactoring: Improving the Design of Existing Code*, Addison-Wesley Professional.

Goel, A. L. and K. Okumoto. 1979. "A time dependent error detection model for software reliability and other performance measures," *IEEE Transactions on Reliability* (28:3), pp. 206-211.

Griswold, W. G. 1991. "Program restructuring as an aid to software maintenance," *University of Washington. Ph.D Thesis*.

Ji, Y., V. S. Mookerjee, et al. 2005. "Optimal Software Development: A Control Theoretic Approach," *Information Systems Research* (16:3), pp. 292–306.

Kemerer, C. F. and S. A. Slaughter. 1997. "Determinants of Software Maintenance Profiles: An Empirical Investigation," *Software maintenance: research and practice* (9:4), pp. 235-251.

Kitchenham, B. A., G. H. Travassos, et al. 1999. "Towards an Ontology of Software Maintenance," *Journal of Software Maintenance: Research and Practice* (11:6), pp. 365-389.

Lientz, B. P. and B. Swanson. 1981. *Software Maintenance Management*, Addison-Wesley

Maruping, L. M., V. Venkatesh, et al 2009. "A Control Theory Perspective on Agile Methodology Use and Changing User Requirements," *Information Systems Research* (20:3), pp. 377-399.

Oman, P., Hagemeister, J. 1992. "Metrics for assessing a software system's maintainability," in *Proceedings of Conference on Software Maintenance*, Orlando, FL , USA, pp. 337-344.

Opdyke, W. 1992. "Refactoring object-oriented frameworks," *University of Illinois at Urbana-Champaign. PhD thesis*.

Poole, C. and J.W.Huisman. 2001. "Using Extreme Programming in a Maintenance Environment," *IEEE Software (*18:6), pp. 42-50.

Poole, C., T. Murphy, et al. 2001. "Extreme maintenance," in *Proceedings of the 17th IEEE International Conference on Software Maintenance, Florence, Italy*.

Rico, D. F. 2008. "Agile Methods and Software Maintenance," *http://davidfrico.com/ricoo8f.pdf*.

Senge, P. 2006. *The Fifth Discipline : The Art & Practice of the Learning Organization*, Doubleday Business.

Shaw, S. 2007. "Using Agile Practices in a Maintenance Environment," *Intelliware Development Inc.*

Svensson, H. and M. Host. 2005. "Introducing an Agile Process in a Software Maintenance and Evolution Organization," *Ninth European Conference on Software Maintenance and Reengineering(CSMR 2005)*.