Measurement of Agility to Change for Service-Based Systems

Kongsawas Potipanom and Twittie Senivongse

***Abstract*—Service orientation has become an attractive solution to many software development and integration problems. To fully adopt the service orientation concept, business organisations need to learn about a new methodology for service architecture and employ it to their existing software systems. Such an effort is likely to pay off later when the service-based systems can accommodate changes in a more flexible way. However, since the benefits are not seen right away, this could deter business organisations, especially smaller enterprises, from service adoption. This paper presents an experiment to determine agility to change of a service-based system in comparison with that of a non-service-based one to check if the service-based version is more agile to change. The two experimental systems are in the context of an E-Leave application of a small software company in Thailand. The model we use to measure agility extends an existing software agility metric by bringing code complexity into the model. The experimental results show that the service-based system is more agile to change than its counterpart, while code complexity can help give a better view of change complexity that affects agility. We also note that agility of the service-based system can be improved if the services are well-designed.**

***Index Terms*—Agility, service orientation, measurement.**

I. Introduction

Literature has addressed the benefits of service orientation to business organisations, including lower time and cost of software development, since business organisations can reuse service building blocks which provide certain functionalities without having to construct those functionalities by themselves [1]. To fully adopt the concept of service orientation, business organisations need to learn about a new analysis and design methodology for service architecture and understand the impact of its employment on their existing software systems. Such an effort is likely to pay off later when the systems can accommodate changes in a more flexible way. However, since these benefits are not seen right away, business organisations, especially smaller enterprises, could be deterred from service adoption.

Agility of a software system refers to the efficiency with which the software system can respond to change requirements [2] and is linked to service orientation since change can be made conveniently to service-based applications, e.g. by adding new services or replacing existing services in the applications with the new ones. We are interested in an experiment to measure agility to change of a service-based system in comparison with that of a non-service-based one in order to determine if service adoption eventually pays off when changes are applied. The two experimental systems are in the context of an E-Leave application of a small software company in Thailand. The model we use to measure agility is an extension of an existing software agility metric which determines agility based on time and the effect of change on software system architecture and on the change process. Extending this model, we also bring complexity of the software code into the picture since the application system may not be so responsive to change if its code is complex.

Section 2 presents research related to different aspects of agility measurement and Section 3 proposes an agility measurement model that is used in our experiment. The methodology of the experiment and the results are discussed in Sections 4 and 5. The paper concludes in Section 6 with the future outlook.

Manuscript received January 14, 2012; revised May 10, 2012.

Kongsawas Potipanom is with the Department of Computer Engineering, Faculty of Engineering Chulalongkorn University, Bangkok, Thailand (e-mail: kongsawas.p@student.chula.ac.th).

Twittie Senivongse is with the Department of Computer Engineering, Faculty of Engineering, Chulalongkorn University, Bangkok, Thailand and also the corresponding author (Tel.: +66 2 2186996; fax: +66 2 2186955; e-mail: twittie.s@chula.ac.th).

II. Related Work

A number of related researches propose different approaches to agility measurement. Most researches focus on process and operation agility such as the work by Arteta and Giachetti [3]. They propose a method for measuring agility of an enterprise through complexity of the Petri Net model of the business process. Another approach by Mansouri et al. [4] proposes an agility index which determines if an interconnected enterprise can adapt to unexpected change without compromising operational time and cost. The work by Datta [5] views agility of a software project based on its characteristics, e.g. project duration, risk, novelty, etc., and determines which development methodology, i.e. Waterfall, UP, and XP, is suitable for the project.

Our focus is more on software agility measurement. Shawky and Ali [6] analyse software change log to determine change frequency of each software entity (e.g. files, classes) at different time periods and associate it with Shannon entropy which is a measure of uncertainty. That is, the higher the rate of entropy change, the more agile the software entity and the development process. However, we see that change frequency of a software entity does not necessarily indicate the ability of the entity itself to respond to change. Rather, the entity may be a frequent target of change if it is defective or does not answer well to user requirements. The approach by Malik [7] sees that a software system with less complexity and less time to

change is more agile. Change complexity is represented by an "area of change", i.e. the smaller the area affected by change, the less the change complexity. The area of change is determined by a product of the number of software architectural layers that are affected by change (i.e. architectural depth) and the number of roles and handoffs between roles within the change process (i.e. process breadth). We follow Malik's intuitive view but also consider code complexity since the application may not be so responsive to change if its code is complex.

III. Agility Measurement Model

We adopt and extend the agility measurement model proposed by Malik in [7] which relates speed of change with complexity of change as in Eq. (1).

Agility = Change Complexity / Change Duration (1)

where

Change Duration = Duration between time of change request and time of change delivery

and

*Change Complexity = Architectural Depth * Process Breadth * Code Complexity* (2)

where *Architectural Depth = number of software architectural layers that are affected by change*

Process Breadth = number of roles in change process + number of handoffs between roles

Code Complexity = McCabe VG complexity of the code to which change will be applied.

Eq. (2) is an extension we make to the original change complexity metric in [7]. As mentioned in Section 2, the original change complexity is represented by size of the "area of change" (i.e. change complexity = architectural depth * process breadth). Here we are interested also in complexity of the code to which change will be applied, i.e. change complexity is "volume of change" as in Fig. 1. From an architectural viewpoint, change can affect different layers of a software system such as user experience, business process, application, and data storage layers. From a change process viewpoint, change may involve different roles within the development team such as analysts, designers, programmers, and testers, and a number of interactions and work products that are passed between team members can signify complexity of change. In addition, we bring code complexity into the picture since behind the architecture and change process lies the application, and making change to application modules can be difficult if the program code is complex. We use McCabe cyclomatic complexity (VG) [8] for code complexity as it is supported largely by software analysis tools.

IV. Methodology of Experiment

To experiment if service orientation can bring agility to software systems, we apply changes to service-based and non-service-based systems and compare their agility using the agility measurement models in [7] and Section 3. The two systems are two versions of an E-Leave application which is a Java Web application that manages leave information of the employees of a small software company in Thailand. The architectural layers of the two systems are shown in Fig. 2. The presentation layer contains UI forms and scripts which are controlled by components in the application layer. The domain layer contains business domain logic, e.g. processing of leave approval and leave taken. The data access layer uses Hibernate framework to facilitate access to EApps leave database. For the service-based version of the application, we follow the service-oriented analysis and design methodology in [1] to introduce services into the system, but only focus on the service categories that are highly reusable, i.e. entity services which provide access to data entities, and utility services which provide utility functions for common use within the system. As a result, the architecture of the service-based version has an additional service layer which comprises three Web services: EmployeeService for access to employee information (entity), EleaveService for access to leave information (entity), and NotificationService for sending notification to employees on leave and approvers (utility).

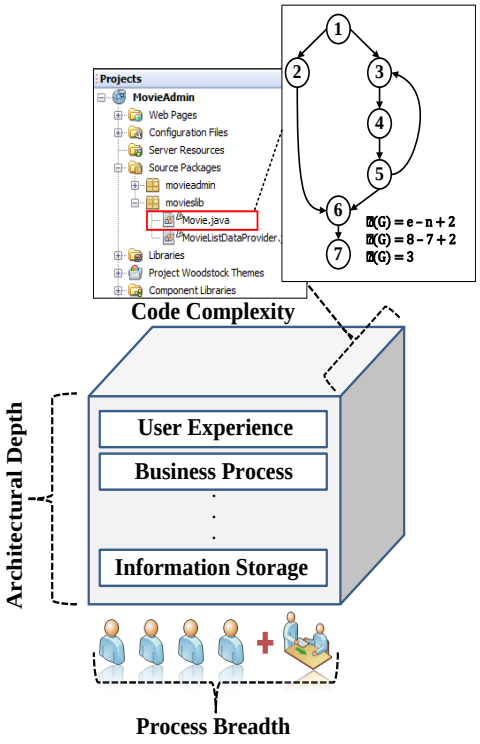

Fig. 1. Volume of change.

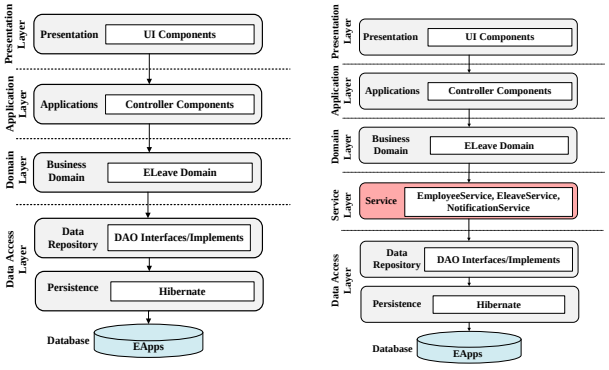

Fig. 2. Architectural layers (a) non-service (b) service.

A. Change Requests

The following change requests (CRs) are applied to the two experimental systems.

- CR01: Add automatic brought forwards of leave to the application: Currently E-Leave does not have the feature to automatically bring forward the leave balance of the present year to the following year. HR staff will have to manually calculate brought forwards and have E-Leave admin staff update the leave balance of the following year for all employees. The rule is no more than half of the leave entitlement for the year can be brought forward from the previous year.
- CR02: Support leave application of shift workers: At present, E-Leave manages leave application of employees who work on weekdays only, i.e. E-Leave does not consider Saturday and Sunday to be leave days even though the leave duration contains them. However, employees who work on shifts can be scheduled to work on weekends, and therefore their leave balance is currently not deducted when they take a leave on the weekends that are scheduled for work. In this case, E-Leave should be able to calculate correctly the number of leave days taken, including leave on weekends.

B. Data Collection

To calculate agility of the two systems, the following data are gathered during the change process.

- Change duration: Timesheets are used to record work duration during processing of change requests.
- Architectural depth: Change sheets are used to record the application modules in different architectural layers that are affected by the change requests.
- Process breadth: Timesheets are also used to record roles in processing change requests and work products that are passed between roles. There are three roles here – system analyst (for analysis and design tasks), programmer, and tester. The same development team works on both systems.
- Code complexity: Prior to applying the changes, a tool called LocMetrics 0 is used to measure McCabe VG complexity of the application modules that will be affected.

V. Results and Discussion

For CR01, timesheets and change sheets for the two versions of E-Leave as well as the summary of the results are shown in Fig. 3. Mainly a single package called com.aoth.eapps in the application layer is changed (Fig. 3(c-d)). For the service-based version, the change is additionally made to EleaveService by adding two operations (Fig. 3(b)), and this makes the number of affected architectural layers and handoffs increase and the complexity of all affected modules higher also (Fig. 3(e)). Nevertheless, change duration is less for the service-based version. Fig. 3(e) also gives a comparison of agility of the two system versions measured by the original agility model [7] and ours as in Section 3. The service-based version is more agile than its counterpart in both cases and the percentage of increase is very similar.

For CR02, timesheets and change sheets for the two system versions as well as the summary of the results are in Fig. 4. More application modules are affected compared to the case of CR01, including three packages – com.aoth.eapps, com.aoth.eapps.domain, and com.aoth.eapps.dto – in the application, domain, and data access layers respectively (Fig. 4(c-d)). For the service-based version, change in the data access layer leads to change in EleaveService and EmployeeService (Fig. 4(b)). Change to these services makes the number of affected architectural layers and handoffs higher than that of the non-service case, and more time is needed to rebuild the system (Fig. 4(e)). Note that complexity of the affected modules is less for the service-based version since the access to the data access layer by the domain layer is facilitated by the service layer and hence the code is less complex. The agility measurements in Fig. 4(e) show that despite the increase in change duration and change complexity, the service-based version is still more agile. The percentage of increase is quite similar for both measurement models although ours is slightly less since we consider code complexity also.

It is seen from the experiment that introducing a service layer can make the software systems more complex from the architectural viewpoint. If the change has to be made in the service layer itself, change complexity will increase, such as the case of CR02 in which change is more major than CR01 and affects many architectural layers and application modules. Change in the service layer may indicate that the services are not designed well according to service design principles, e.g. reusability [2]. Services may not be generic enough to be used in different contexts and hence have to be modified to add new operations or to change logic and data behind the services when there are new requirements. This is often the case for service design especially in small business organisations which tend to build services with limited time and resources and for a particular purpose. Nevertheless, agility measurements from our model and the original model agree that the service concept can improve agility to change. The code complexity factor that we consider helps give more insight into the application system; it is seen from the CR02 case that the use of several services in the application can clearly decrease the overall complexity of the code.

No.	Phase	Description	Role	Timesheet Date	Working Hour	Remark	Work Product
1	Design	Use Case Diagram, Class Diagram, Activity Diagram	System Analyst	9-Dec-11	3		
2	Design	Design Sequence Diagram	System Analyst	10-Dec-11	3		Design Specification
3	Implement	Coding	Programmer	10-Dec-11	8	Create Leave Brought Forward Screen	
4	Implement	Coding	Programmer	11-Dec-11	8	Create Leave Brought Forward Screen and Implement Logic	
5	Implement	Coding & Deploy Application	Programmer	12-Dec-11	4	Implement Logic	E-Leave Application
6	Testing	Functional Test	Tester	13-Dec-11	3		

(a)

Total Working Hour: 29

No.	Phase	Description	Role	Timesheet Date	Working Hour	Remark	Work Product
1	Design	Use Case Diagram, Class Diagram, Activity Diagram	System Analyst	14-Dec-11	4		
2	Design	Design Sequence Diagram	System Analyst	15-Dec-11	3		Design Specification
3	Implement	Coding	Programmer	16-Dec-11	4	Create Leave Brought Forward Screen	
4	Implement	Coding	Programmer	17-Dec-11	8	- Implement Logic - Add Method *batchAddLeaveAdjustment()* and *batchUpdateLeaveAdjustment()* *- Re-deploy Service*	EleaveService
5	Implement	Coding & Deploy Application	Programmer	18-Dec-11	4	Implement Logic	E-Leave Application
6	Testing	Functional Test	Tester	18-Dec-11	4		

(b)

Total Working Hour: 27

Layer (Non-Service)	Affected JSP Files/Packages/Java Classes
Presentation	ELeaveBringForward.jsp
Application	com.aoth.eapps, ELeaveBringForward.java
Domain	-
Data Access	-

(c)

Layer (Service)	Affected JSP Files/Packages/Java Classes/Services
Presentation	ELeaveBringForward.jsp
Application	com.aoth.eapps, ELeaveBringForward.java
Domain	-
Service	EleaveService
Data Access	-

(d)

E-Leave System	Duration (hrs.)	Arch. Depth (no.)	Roles (no.)	Handoffs (no.)	Code Complexity (MVG)	Agility [7]	Agility Increase [7] (%)	Our Agility	Our Agility Increase (%)
Non-Service	29	2	3	2	2,298	0.34	94.12	792.41	94.17
Service	27	3	3	3	2,308	0.66		1,538.66	

(e)

Fig. 3. CR01 results: Timesheet (a) non-service (b) service; Change sheet (c) non-service (d) service; (e) Summary.

No.	Phase	Description	Role	Timesheet Date	Working Hour	Remark	Work Product
1	Design	Design Use Case Diagram, Class Diagram	System Analyst	26-Dec-11	2		
2	Design	Design Activity Diagram, Sequence Diagram	System Analyst	27-Dec-11	3		Design Specification
3	Implement	Coding	Programmer	28-Dec-11	4	- Update Table El_Employee, El_Leave_Record - Update DTO "ElEmployee", "ElLeaveRecord" - Update "Update Employee Info" Screen	
4	Implement	Coding	Programmer	29-Dec-11	4	- Update "Update Employee Info" Screen	
5	Implement	Coding	Programmer	3-Jan-12	4	- Update "Apply Leave" Screen and E-Leave Domain	
6	Implement	Coding & Deploy Application	Programmer	4-Jan-12	4	- Update "Apply Leave" Screen	E-Leave Application
7	Testing	Functional Test	Tester	5-Jan-12	4		

(a)

Total Working Hour: 25

	No.	Phase	Description	Role	Timesheet Date	Working Hour	Remark	Work Product
	1	Design	Use Case Diagram, Class Diagram, Activity Diagram	System Analyst	6-Jan-12	4		
	2	Design	Design Sequence Diagram	System Analyst	7-Jan-12	2		Design Specification
	3	Implement	Coding	Programmer	7-Jan-12	4	- Update Table El_Employee, El_Leave_Record - Update DTO - Re-deploy EmployeeService, EleaveService	EmployeeService, EleaveService
(b)	4	Implement	Coding	Programmer	8-Jan-12	10	- Update "Update Employee Info" Screen - Update "Apply Leave" Screen and E-Leave Domain	
	5	Implement	Coding & Deploy Application	Programmer	9-Jan-12	4	Update "Apply Leave" Screen and E-Leave Domain	E-Leave Application
	6	Testing	Functional Test	Tester	10-Jan-12	4		
					Total Working Hour:	**28**		

Layer (Non-Service)	Affected JSP Files/Packages/Java Classes
Presentation	EleaveNewUpdateEmployee.jsp
Application	com.aoth.eapps, EleaveNewUpdateEmployee.java, ELeaveApplyLeave.java
Domain	com.aoth.eapps.domain, ELeaveDomain.java,
Data Access	com.aoth.eapps.dto, ElEmployee.java, ElLeaveRecord.java

(c)

Layer (Service)	Affected JSP Files/Packages/Java Classes/Services
Presentation	EleaveNewUpdateEmployee.jsp
Application	com.aoth.eapps, EleaveNewUpdateEmployee.java, ELeaveApplyLeave.java
Domain	com.aoth.eapps.domain, ELeaveDomain.java,
Service	EleaveService, EmployeeService
Data Access	com.aoth.eapps.dto, ElEmployee.java, ElLeaveRecord.java

(d)

E-Leave System	Duration (hrs.)	Arch. Depth (no.)	Roles (no.)	Handoffs (no.)	Code Complexity (MVG)	Agility [7]	Agility Increase [7] (%)	Our Agility	Our Agility Increase (%)
Non-Service	25	4	3	2	2,746	0.8	56.25	2,196.80	55.23
Service	28	5	3	4	2,728	1.25		3,410.00	

(e)

Fig. 4. CR02 results: Timesheet (a) non-service (b) service; Change sheet (c) non-service (d) service; (e) Summary

VI. Conclusion

This paper presents an experiment on measuring agility of a non-service-based application system and its service-based counterpart. The measurement is based on speed of change and complexity of change. A comparison is given between an existing agility model [7] and our extension which adds code complexity to the change complexity dimension. The measurements from our model match those of the original model and indicate that the service-based version is more agile. In addition, agility can be improved even better with good service design.

This experiment is merely an example of the benefit of service orientation to agility and considers a number of factors that can contribute to agility in the measurement model. To formally confirm the idea, a statistical approach should be taken. That is, the experiment should involve more application systems and different kinds of change. We can test the correlation between agility and different factors of the application and the change, and check to see if code complexity has statistical significance in the measurement model. We can also improve the design of the service-based version and see if its agility also improves.

References

[1] T. Erl, *Service-Oriented Architecture: Concepts, Technology, and Design*, Prentice Hall, 2005.

[2] T. Erl, *SOA Principles of Service Design*, Prentice Hall, 2008.

[3] B. M. Arteta and R. E. Giachetti, "A measure of agility as the complexity of the enterprise system," *Robotics and Computer-Integrated Manufacturing*, Science Direct, vol. 20, pp. 495-503, 2004.

[4] M. Mansouri, A. Ganguly, and A. Mostashari, "Evaluating Agility in Extended Enterprise Systems: A Transportation Network case," *American J. of Engineering and Applied Sciences.* Science Publications, vol. 4, no. 1, pp. 142-152, 2011.

[5] S. Datta, "Agility Measurement Index – A Metric for the Crossroads of Software Development Methodologies," in *Proc. of 44th Annual Southeast Regional Conference (ACM-SE 44)*, New York: ACM. 2006, pp. 271-273.

[6] D. M. Shawky and A. F. Ali, "A Practical Measure for the Agility of Software Development Processes," in *IEEE Proc. of 2nd Int. Conf. on Computer Technology and Development (ICCTD 2010*), 2010, pp. 230-234.

[7] N. Malik. (14 December 2007). Measuring the agility of a SOA approach. [Online]. Available: http://blogs.msdn.com/b/nickmalik/archive/2007/12/14/measuring-the-agility-of-a-soa-approach.aspx

[8] T. J. McCabe, "A Complexity Measure," *IEEE Trans. on Software Engineering*, SE-2(4), pp. 308-320, 1976.

LocMetrics.[Online] Available: http://www.locmetrics.com/..