Towards Facilitating Development of SOA Application with Design Metrics

Wei Zhao, Ying Liu, Jun Zhu, and Hui Su

IBM China Research Lab,
Beijing 100094, P.R. China
{weizhao, aliceliu, junzhu, suhui}@cn.ibm.com

Abstract. Applications based on service-oriented architecture (SOA) are intended to be built with both high cohesion and low coupling. The loosely coupled services bring forth the lower costs of development and maintenance as well as the higher reusability and extensibility. To implement each SOA application with such intention, designs play an important role for the success of the whole project. The services and the relationships among them represented in a design are two critical factors to decide the quality of an SOA application in terms of modularity. At the mean while, they are valuable indicators for guiding the following development and maintenance phases to progress in a cost-effective way. In this paper, we present that measurement of designs for SOA applications can objectively judge the quality and further facilitate the development and maintenance of SOA applications through employing two specific metrics. We also performed an experimental study on an ongoing SOA project. In this study, we applied these two metrics to the design of this project to acquire judgments and make estimations. The data in CVS were retrieved to reflect the genuine project situations. The analysis on these data shows that adopting the measurement in the early stage of SOA projects may avoid wasting efforts and delaying schedule as well as acquire a deep grasp and an effective control on the issues in the following phases.

Keywords: SOA, modular design, service design measurement, metrics.

1 Introduction

Modular architectures solve the problem of complexity of the business by decomposing complex services into modules so that service providers can design and deliver them independently [1]. These business service modules are implemented by the corresponding service modules and components (within the implementation context, service components are programming entities with finer granularity than service modules) at IT level under the service-oriented architecture (SOA) and the supporting programming model (e.g. service component architecture, SCA) as an SOA application. The descriptions of which service modules and components construct an SOA application and how they are interrelated to provide the business services are regarded as the architecture design of an SOA application. Accordingly, these service modules and components at IT level should hold the similar modular properties in finer granularity to satisfy the modular design of business services. That is to say, SOA applications consisting of various service modules and components are intended to be built by showing both high cohesion and low coupling [2]. The architecture of an SOA application with higher cohesion as well as lower coupling indicates a better design in terms of modularity. A well-modularized design of an SOA application brings forth potential benefits in multiple aspects, such as acceleration of development, reduction of maintenance cost, as well as the enhanced flexibility and reusability.

The quality of the modular designs for SOA applications often heavily relies on the experiences and expertise of specific designers. In addition, the best practices and design patterns as well as frameworks, which are summarized from accumulated experiences, can be a useful guidance for a better design. However, these facilities are still kinds of informal aids to modular design and the achieved effectiveness from them still heavily depends on the experiences and expertise of individual designers to some extent.

There are no any practical reports in industry to employ measurement technologies to evaluate whether a certain design of an SOA application is well modularized than another or to guide the activities in the following development and maintenance phases through design metrics. Actually, to employ measurement to judge the modularity of software designs is not new. Many efforts have been dedicated to judge and reorganize the structural designs of software systems according to the modularized degrees (e.g. [3] and [4]). However, with the intention to acquire the loosely coupled SOA applications, service-oriented architecture does provide a framework to model the constructive entities (i.e. interfaces, service components, and service data objects) and their interrelationships more explicitly at a higher abstract level, but it does not mean that any application based on SOA holds the loose coupling and tight cohesion inherently.

In this paper, we report an initial exploration of measurement on SOA designs. We present that measurement of designs for SOA applications can quantitatively evaluate the quality of modular designs through a comparative way and also can facilitate the development and maintenance of applications.

We performed an experimental study on an ongoing SOA project. In this study, we employed two metrics on the design of this project to acquire judgments and make estimations. The corresponding data in CVS were retrieved to reflect the genuine project situations. The analysis on these data shows that adopting the design metrics in the early stage of SOA projects may avoid wasting efforts and delaying the schedule as well as acquire an early grasp and effective control on the issues in the following phases.

The remainder of this paper is organized as follows. Section 2 introduces the goals we want to achieve through measuring designs for SOA applications and the corresponding design metrics we used. An experimental study on an ongoing SOA application is presented in section 3 to validate the effectiveness of the metrics and imply their indicating and aiding roles. Section 4 summarizes this paper.

2 Goals and Design Metrics

2.1 Goals

A measurement program identifies and defines metrics to support an organization's business goals [5]. These metrics provide insights into the critical quality and management issues that the organization concerns for its success. During the establishment of a measurement program, the organization selects metrics traceable to its business goals. This "goal-driven" approach assures that measurement activities stay focused on the organization's objectives.

Because the well modularized designs of SOA applications bring multiple benefits such as reducing the development and maintenance cost and increasing the reusability as mentioned above, one of our goals is to quantify the modular designs of SOA applications in terms of the estimated relative development cost and maintenance cost in an early stage (i.e. right after acquiring the designs of applications). The design metrics in this paper refer to these quantitatively estimated indicators for the costs of following development and maintenance activities based on the design information. Although we aim to acquire the quantitative insights on how well a modular design is, it should be noted that we examine such merit through a comparative way. That is to say, we cannot claim that a specific SOA application is well designed enough in terms of modularity even with the quantitative metrics. However, given the two candidate designs for a certain application, we can quantitatively judge that one is better (or worse) than the other in terms of modularity and make a choice for lower development and maintenance costs.

In addition to the quantitative evaluation of the whole design, the comparison based on the design metrics can also be carried out within a specific SOA application design to pinpoint the modular characteristics of each service module and component. For a determined design of an SOA application, further scrutinizing each service module and component based on the design metrics provides the valuable insights to the following development and maintenance phases. This is the other goal we expect to pursue through the design metrics.

2.2 Metrics Definition

We adopt a technique called Design Structure Matrix (DSM) [6] to analyze the designs of SOA applications. A DSM is a tool that highlights the inherent structure of a design by examining the dependencies that exist between its component elements using a symmetric matrix.

The component elements in the design of an SOA application are service components. As service modules are composed of service components, the metrics of a service module can be acquired through calculating the service components belonging to it. As a result, the service modules' corresponding metrics will not be omitted although they are not explicitly represented in the design structure matrix. To construct the design structure matrix based on the service components of an SOA application's design, we follow Parnas's "information hiding" criterion [7] to mark the dependencies among the service components which are further used to measure and judge the modularity of an SOA application's design. In more detail, for the design of an SOA application, each service component may operate (i.e. *create*, *update*, *read* and *delete*) some data objects. Due to the dependent operations on the same data objects, the service components are interrelated among others. These dependencies are the key factors to identify how well the investigated design is modularized according to the "information hiding" principle from the perspective of the operated data.

Based on the constructed design structure matrix presented above, we employ two DSM-based metrics originally proposed by MacCormack et. al. to estimate the phenomena with which the design structure of software are associated [3]. MacCormack et. al.'s work focuses on the predication through an overall design to compare the modularity of two candidate designs. Since we aim at not only providing a quantitative cognition of a current modular design but also facilitating the subsequent development and maintenance activities, we further employ these two metrics to scrutinize the designs in the finer granularity. The definitions of these two metrics, change cost and coordination cost, are introduced as follows:

Change Cost

Change cost is a metric which determines the impact of a change to each service component, in terms of the percentage of other service components that are potentially affected. It is an indicator of the efforts needed for the maintenance activities of an SOA project. Obviously, the higher the change cost, the worse the design is modularized. In MacCormack et. al.'s work, this metric is computed only for the overall design of software. We also scrutinize this metric for each service component of a specific design. Actually, change cost of the overall design is an average of all service components' change costs.

	A	***B***	***C***	***D***	***E***	***F***
A	0	1	1	0	0	0
B	0	0	0	1	0	0
C	0	0	0	0	1	0
D	0	0	0	0	0	0
E	0	0	0	0	0	1
F	0	0	0	0	0	0

Fig. 1. An example design structure matrix

To acquire the change cost of a service component or an SOA application (change cost of a service module consisting of service components can be acquired through the same way as the SOA application), a matrix is constructed firstly to represent the structure of the design of an SOA application as described above. If an SOA application is composed of n service components, the size of the matrix is $n \times n$. Each cell of this matrix indicates the modular dependency between the service components in the corresponding column and row based on the "information hiding" principle from the perspective of the operated data as mentioned above.

The computation of change cost is illustrated by the following example. Considering the relationships among service components displayed in the design structure matrix in Fig. 1, it can be seen that *service component A* depends on *service* *component B* and *C*. Therefore any change to *service component B* may have a direct impact on *service component A*. Similarly, *service component B* and *C* depend on *service component D* and *service component E* respectively. Consequently, any change to *service component D* may have a direct impact on *service component B*, and may have an indirect impact on *service component A*, with a "path length" of 2.

Obviously, the technique of matrix multiplication can be used to identify the impacted scope of any given service component for any given path length. Specifically, by raising the matrix to successive powers of *n*, the results show the direct and indirect dependencies that exist for successive path lengths. By summing these matrices together, the matrix *V* (which is called as the visibility matrix) can be derived, showing the dependencies that exist for all possible path lengths up to *n*. It should be noted that this calculating process includes the matrix for *n*=0 (i.e., a path length of zero) when calculating the visibility matrix, implying that a change to a service component will always affect itself. Fig. 2 illustrates the calculation of the visibility matrix for the above example.

M0	*A*	*B*	*C*	*D*	*E*	*F*	M1	*A*	*B*	*C*	*D*	*E*	*F*	M2	*A*	*B*	*C*	*D*	*E*	*F*
A	1	0	0	0	0	0	***A***	0	1	1	0	0	0	***A***	0	0	0	1	1	0
B	0	1	0	0	0	0	***B***	0	0	0	1	0	0	***B***	0	0	0	0	0	0
C	0	0	1	0	0	0	***C***	0	0	0	0	1	0	***C***	0	0	0	0	0	1
D	0	0	0	1	0	0	***D***	0	0	0	0	0	0	***D***	0	0	0	0	0	0
E	0	0	0	0	1	0	***E***	0	0	0	0	0	1	***E***	0	0	0	0	0	0
F	0	0	0	0	0	1	***F***	0	0	0	0	0	0	***F***	0	0	0	0	0	0
M3	***A***	***B***	***C***	***D***	***E***	***F***	**M4**	***A***	***B***	***C***	***D***	***E***	***F***	**V**	***A***	***B***	***C***	***D***	***E***	***F***
A	0	0	0	0	0	1	***A***	0	0	0	0	0	0	***A***	1	1	1	1	1	1
B	0	0	0	0	0	0	***B***	0	0	0	0	0	0	***B***	0	1	0	1	0	0
C	0	0	0	0	0	0	***C***	0	0	0	0	0	0	***C***	0	0	1	0	1	1
D	0	0	0	0	0	0	***D***	0	0	0	0	0	0	***D***	0	0	0	1	0	0
E	0	0	0	0	0	0	***E***	0	0	0	0	0	0	***E***	0	0	0	0	1	1
F	0	0	0	0	0	0	***F***	0	0	0	0	0	0	***F***	0	0	0	0	0	1

Fig. 2. Successive powers of the design structure matrix and visibility matrix

From the visibility matrix, the change cost metric can be acquired to give the insight for each service component and the whole SOA application. Firstly, for each service component, the change cost is obtained by summing along the column of the visibility matrix, and dividing the result by the total number of service components. A service component with higher change cost possibly affects more service components while changing it. In the above example, *service component F* has a change cost of 4/6 (or 66.67%) which means a change on it may affect other 4 service components in the system.

The average change cost of all service components need to be computed for the whole design. The resulting metric is the change cost for the overall design of a given SOA application. Intuitively, this metric reflects the percentage of service components affected on average when a change is made to a certain service component in the application. In the example above, we can calculate the change cost of the overall design as [1/6+2/6+2/6+3/6+3/6+4/6] divided by 6 service components = 41.67%.

Coordination Cost

Coordination cost is a metric to evaluate how well the proposed design of an SOA application in terms of the coordinating efforts needed in the procedure of developing it in the future. It is an indicator of the efforts needed for the development activities of an SOA project. The higher the coordination cost, the worse the design is modularized. Different from the change cost, the coordination cost is not only determined by the dependencies between the constructing service components but also affected by how these service components are organized into different service modules. The calculation of coordination cost metric operates by allocating a cost to each dependency between service components firstly. Specifically, for an SOA application, when considering a dependency between service components *A* and *B*, the cost of the dependency takes one of following two forms:

$$CoordCost\ (A \rightarrow B|\text{in same module}) = (A \rightarrow B)^{cost\_dep} \times size\ of\ module^{cost\_cs} \quad (1)$$

$$CoordCost\ (A \rightarrow B|\text{not in same module}) = (A \rightarrow B)^{cost\_dep} \times sum\ size\ of\ two\ modules^{cost\_cs} \quad (2)$$

Where (*A*→*B*) represents the strength of the dependency (that is, the number of correlations between *service component A* and *B*) and *cost_dep* and *cost_cs* are user-defined parameters that reflect the relative weights given to the strength of dependencies versus the size of the modules.

For each service component, the corresponding coordination cost is determined through summing up all *CoordCost* between it and all its dependent service components. The coordination cost of the overall design of an SOA application can be acquired from summarizing the coordination costs of all the service components in the design.

3 Experimental Study

3.1 Experimental Method

Rifkin and Cox performed case studies on software measurement programs of different corporations and reported that the most successful programs they observed supported experimentation and innovation [8]. Following the similar point of view, we performed a pilot experimental study to validate effects of adopted metrics and initiate the measurement program on the designs of SOA applications for some particular project goals.

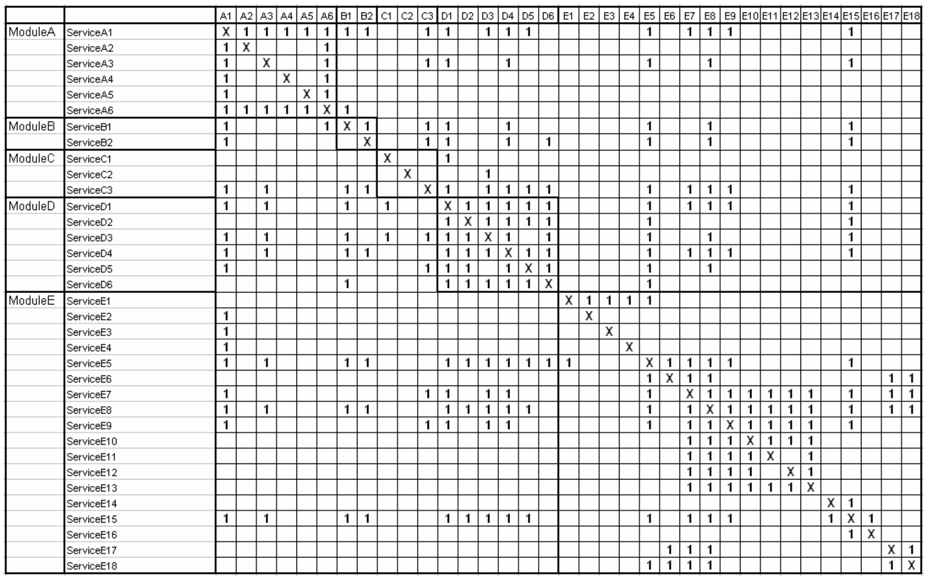
The subject system in our study is an SOA project as a proof of concept for early convincing the customers. The specific requirements on this project include implementing the basic functionalities as customer needed within a short time as well as low cost. Although the scope of this project is not big enough as a real SOA project, it does represent the key factors and characteristics of an SOA application. It should be noted that we did the experimental study not through applying the design metrics we introduced above to guide the development and maintenance activities of this project. We adopted a way using the project data without affected by the design metrics to provide the evidences whether the design metrics make the right estimates and whether the estimates can provide the effective advices for the following stages to help achieve the goals of this project.

3.2 Data Analysis and Observations

The subject SOA project is composed of five service modules each of which are further implemented by service components. Due to the confidential consideration, we use *ModuleA*, *ModuleB*, *ModuleC*, *ModuleD* and *ModuleE* designating these five service modules. An overall implementation situation of the subject system is listed in Table 1. As we can see, *ModuleA* includes 6 service components which provide services (each service component implements one service) to be consumed by end users directly or by other services. Each service component is implemented by an entrance class as well as other related classes. All these 6 services have 11 operations to perform the specific tasks provided by these services. Methods of entrance class correspond to each service's operations. *ModuleA* is implemented by 13 Java files including 6 entrance classes for 6 services respectively and 7 related classes (We do not further include those supporting classes since they are not the interferential factors for the analysis).

Table 1. Modules, services, operations and Java files of subject system

Modules	Services	Operations	Java files
ModuleA	6	11	13
ModuleB	2	4	10
ModuleC	3	10	3
ModuleD	6	13	33
ModuleE	18	26	47



		A1	A2	A3	A4	A5	A6	B1	B2	C1	C2	C3	D1	D2	D3	D4	D5	D6	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10	E11	E12	E13	E14	E15	E16	E17	E18
ModuleA	ServiceA1	X	1	1	1	1	1	1	1			1	1		1	1	1						1		1	1	1						1			
	ServiceA2	1	X				1																													
	ServiceA3	1		X			1					1	1			1							1			1							1			
	ServiceA4	1			X		1																													
	ServiceA5	1				X	1																													
	ServiceA6	1	1	1	1	1	X	1																												
ModuleB	ServiceB1	1					1	X	1			1	1			1							1			1							1			
	ServiceB2	1							X			1	1			1		1					1			1							1			
ModuleC	ServiceC1									X			1																							
	ServiceC2										X				1																					
	ServiceC3	1		1				1	1			X	1		1	1	1	1					1		1	1	1						1			
ModuleD	ServiceD1	1		1				1		1			X	1	1	1	1	1					1		1	1	1						1			
	ServiceD2												1	X	1	1	1	1					1										1			
	ServiceD3	1		1				1		1		1	1	1	X	1		1					1			1							1			
	ServiceD4	1		1				1	1				1	1	1	X	1	1					1		1	1	1						1			
	ServiceD5	1										1	1	1		1	X	1					1			1										
	ServiceD6							1					1	1	1	1	1	X					1													
ModuleE	ServiceE1																		X	1	1	1	1													
	ServiceE2	1																		X																
	ServiceE3	1																			X															
	ServiceE4	1																				X														
	ServiceE5	1		1				1	1				1	1	1	1	1	1	1				X	1	1	1	1						1			
	ServiceE6																						1	X	1	1									1	1
	ServiceE7	1										1	1		1	1							1		X	1	1	1	1	1	1		1		1	1
	ServiceE8	1		1				1	1				1	1	1	1	1						1		1	X	1	1	1	1	1		1		1	1
	ServiceE9	1										1	1		1	1							1		1	1	X	1	1	1	1		1			
	ServiceE10																								1	1	1	X	1	1	1					
	ServiceE11																								1	1	1	1	X		1					
	ServiceE12																								1	1	1	1		X	1					
	ServiceE13																								1	1	1	1	1	1	X					
	ServiceE14																															X	1			
	ServiceE15	1		1				1	1				1	1	1	1	1						1		1	1	1					1	X	1		
	ServiceE16																																1	X		
	ServiceE17																							1	1	1									X	1
	ServiceE18																						1	1	1	1									1	X

Fig. 3. Design structure matrix of subject system

According to the descriptions of the construction of the design structure matrix in section 2, we acquired the DSM of the subject system which can be seen in Fig. 3. The dependencies among the service components of the subject system were picked out and filled in the matrix through the analysis on the dependent operations on data

Table 2. Change cost and coordination cost of service components and the overall system

Service components	**Change cost (%)**	**Coordination cost**
ServiceA1	60	229
ServiceA2	9	18
ServiceA3	29	123
ServiceA4	9	18
ServiceA5	9	18
ServiceA6	20	44
ServiceB1	31	101
ServiceB2	23	99
ServiceC1	9	12
ServiceC2	3	12
ServiceC3	26	181
ServiceD1	49	197
ServiceD2	26	84
ServiceD3	37	152
ServiceD4	46	196
ServiceD5	29	99
ServiceD6	26	68
ServiceE1	6	90
ServiceE2	6	42
ServiceE3	6	42
ServiceE4	6	42
ServiceE5	54	358
ServiceE6	11	108
ServiceE7	46	315
ServiceE8	6	406
ServiceE9	37	279
ServiceE10	20	126
ServiceE11	17	108
ServiceE12	17	108
ServiceE13	20	126
ServiceE14	6	36
ServiceE15	46	334
ServiceE16	6	36
ServiceE17	14	90
ServiceE18	14	108
Overall	24	4405

objects based on the "information hiding" principle. It can be seen from this figure that *ServiceA1* of *ModuleA* depends on *ServiceA2*, *A3*, *A4*, *A5* and *A6* of *ModuleA*, *ServiceB1*, *B2* of *ModuleB*, *ServiceC3* of *ModuleC*, *ServiceD1*, *D3*, *D4* and *D5* of *ModuleD*, and *ServiceE5*, *E7*, *E8*, *E9* and *E15* of *ModuleE*. According to the definitions of two metrics presented in section 2.2, change costs and coordination costs of the overall SOA application and each service component in this application were acquired correspondingly in Table 2. The change cost of the overall SOA application is 24% and the coordination cost is 4405. We assigned the value "1" to the *cost_dep* and *cost_cs* just for the simplicity of the calculation. Although we do not have another candidate design of the subject system as a counterpart to validate the metrics for the overall design, the following analysis based on each service component does validate the metrics and present the potential spaces where the measurement of design could help for the development and maintenance.

As introduced above, *ModuleA* includes 6 service components which implement 6 services. Fig. 4 shows the development data of each service component in *ModuleA* acquired from the project's CVS database, where Axis-X indicates the working days passed while the project progresses and Axis-Y indicates the working efforts consumed until a particular working day. We simply use the lines of code (LOC) to denote the working efforts since the subject system of our experimental study was at its initial stage and the complexity of components does not affect much on working efforts. As a result, in Fig. 4, each service component in *ModuleA* has a corresponding pillar when its implementation source code was checked in the CVS database. The height of a pillar means how many lines of code have been added, deleted or modified since the beginning rather than the lines of code of current source files checked in for each service component.

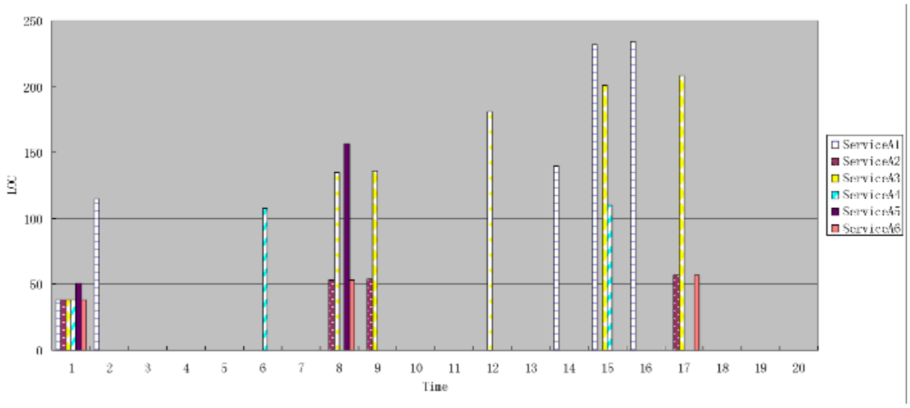


Fig. 4. Development data of ModuleA acquired from CVS

Combining the spent working days and working efforts we can acquire the cognition of the development cost and the schedule of each service component in *ModuleA*. As we can see that the implementations of all service components in *ModuleA* began at the same day. Except *ServiceA1* and *ServiceA3*, the other service

Table 3. Sizes of service components in ModuleA

Services in ModuleA	Size (LOC)
ServiceA1	213
ServiceA2	54
ServiceA3	134
ServiceA4	110
ServiceA5	157
ServiceA6	54

components in *ModuleA* finished the initial versions in 9 days. The working efforts on *ServiceA4* at the 15th day as well as *ServiceA2* and *ServiceA6* at the 17th day were due to fixing the bugs discovered through the integration testing. *ServiceA1* and *ServiceA3* spent 6 more days than others to accomplish their initial versions at the 15th day. Although the size of *ServiceA1* (the lines of code of the finally implemented service component) is larger than other service components as shown in Table 3, the differentiation at such order of magnitude is not a critical factor for the additional six working days. Moreover, the size of *ServiceA3* is even less than *ServiceA5*, but it still costs more. Actually, such situation was caused by the average assignment of resources for each service component since the different working efforts were not carefully taken into considered for the schedule. However, as we can see from the acquired metrics of service components in *ModuleA* in Table 2, due to *ServiceA1* and *ServiceA3* hold the dependencies to the service components in all the other modules (which can be seen in Fig. 3), the implementation of *ServiceA1* and *ServiceA3* has to coordinate with the implementation of other service components and therefore the higher development cost for *ServiceA1* and *ServiceA3* (229, 123 respectively) can be estimated through the design information. Consequently, through the above analysis, we firstly validate the effectiveness of the coordination cost metric. It does present a correct estimation of development cost in early stage. Such early indication can help service designers discover the problems of current service modular design in time. In addition, in case that the dependencies between services can not be easily resolved due to some constraints, we also state that the comparative analysis of coordination costs of the service components can help acquire a reasonable and cost-effective resource assignment and working schedule. If the coordination cost was taken into consideration right after the design was acquired, *ServiceA1* and *ServiceA3* would be assigned more resources than other service components to avoid wasting the efforts due to simply average assignment as well as to shorten the working days.

Continuing to investigate the development data of *ModuleD* in Fig. 5, the effectiveness and merits of change cost metric can be further discovered. After finishing the first version of *ModuleD* (at the 17th working day), there was a change request to *ServiceD4* from customers. However, the individual developers did not know that *ServiceD1* and *ServiceD3* depend on *ServiceD4* and the change was performed to *ServiceD4* only at first. The modifications on *ServiceD1* and *D3* were only accomplished two days later triggered by the testing. From the acquired change cost metric in Table 2, potential change impacts are estimated for each service component. Although current project data does not provide the proof to validate this metric through comparing the costs due to changes on two different service

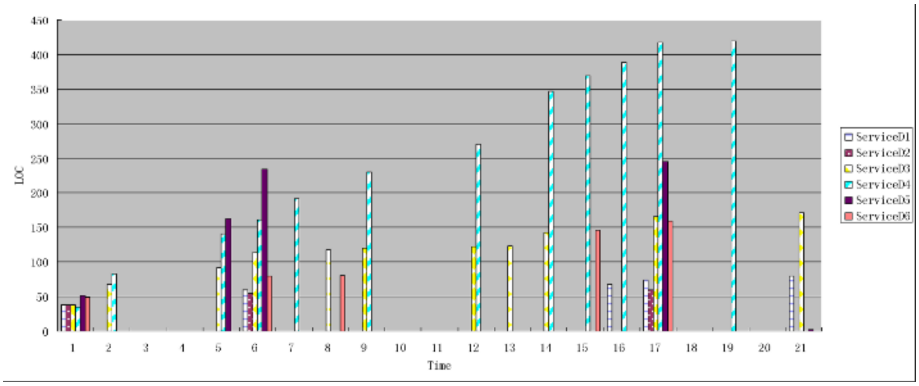


Fig. 5. Development data of ModuleD acquired from CVS

components, change cost metric does provide a quantitative and conservative estimate for potentially affected service components according to the case of a change request to *ServiceD1*. It is obvious that such metric is an effective aid for the maintenance activities. Besides, we can further acquire the specific service components potentially affected through checking the service design structure and therefore provide the effective guidance for specific change requests to service components.

4 Summary

In this paper, we present that measurement of the designs for SOA applications can evaluate the quality of modularity and facilitate the development and maintenance of SOA applications with high efficiency. We performed an experimental study on an ongoing SOA project. In this study, we employed two metrics (change cost and coordination cost) on the design of this project to acquire judgments and make estimations. The project data in CVS was retrieved to reflect the genuine situations of its implementation, integration and testing. The analysis on these data shows that adopting the design metrics in early stage of SOA projects may avoid wasting efforts and delaying the schedule as well as acquire a deep grasp and effective control on the issues in following phases.

References

1. Bohmann, T. and Loser, K.U.: Towards a service agility assessment - Modeling the composition and coupling of modular business services. In Proceedings of the 7th IEEE International Conference on E-Commerce Technology Workshops, 2005: 140-148.
2. Brocke, J. and Lindner, M.A.: Service portfolio measurement: a framework for evaluating the financial consequences of out-tasking decisions. In Proceedings of 2nd International Conference on Service-Oriented Computing, November 15-19, 2004: 203-211.
3. MacCormack, A., Rusnak, J. and Baldwin, C.: Exploring the structure of complex software designs: an empirical study of open source and proprietary code. Harvard Business School Working Paper, Number 05-016, 2004.

4. Schwanke, Robert W.: An intelligent tool for re-engineering software modularity. In Proceedings of the 13th International Conference on Software Engineering. Washington, DC: IEEE Computer Society Press, May 1991: 83-92.
5. Park, R.E., Goethert, W.B., and Florac, W.A.: Goal-driven software measurement - a guidebook. (CMU/SEI-96-HB-002, ADA313946). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1996. http://www.sei.cmu.edu/publications /documents/96.reports/96.hb.002.html
6. Steward, Donald V.: The design structure system: a method for managing the design of complex systems. IEEE Transactions on Engineering Management, vol. 28, pp. 71-74, 1981.
7. Parnas, D.: On the criteria to be used in decomposing system into modules. Communications of the ACM, 15(12):1053-1058, December 1972.
8. Rifkin, S. and Cox, C.: Measurement in practice. Technical report of CMU SEI, TR16.91, July, 1991.