# ESTIMATING PATTERNS CONSEQUENCES FOR THE ARCHITECTURAL DESIGN OF E-BUSINESS APPLICATIONS

Feras T. Dabous[1], Fethi A. Rabhi[1], Hairong Yu[1], and Tariq Al-Naeem[2]
[1]*School of Information Systems, Technology and Management*
[2]*School of Computer Science and Engineering*
*The University of New South Wales, 2052 Sydney NSW, Australia*

Keywords: Patterns, Legacy Systems, Development Methodology, e-Business Applications, e-Finance.

Abstract: Quality attributes estimations at early stages of the design process play an important role in the success of e-business applications that support the automation of essential Business Processes (BPs). In many domains, these applications may utilise functionalities that are embedded in a number of monolithic and heterogeneous legacy systems. In previous work, we have identified a range of patterns that capture best practices for the architectural design of such applications with the presence of legacy functionality. In this paper, we present and discuss quantitative patterns' consequences models to systematically estimate two quality attributes that are the development and maintenance efforts. A real life case study in the domain of e-finance and in particular capital markets trading is used in this paper to validate these models.

## 1 INTRODUCTION

The concept of e-business applications has emerged as an acronym of distributed applications that utilize the Internet as a medium for coordinating the interactions among different distributed components. Each component may correspond to an activity that implements part of an organisations business logic. In this research, we consider e-business application domains that can be improved by developing or utilising corresponding business logic and functionality that may span across different organisational legacy systems. In many domains such as e-finance, these applications are business process intensive and therefore we alternatively use the term Business Process (BP) to correspond to an e-business application.

Legacy systems are valuable assets within organisations many of which are expensive, reliable and efficient in supporting businesses. As these systems evolve, they become complex and hard to adapt to new business requirements (Umar, 1997). However, such systems can play an important role in automating new BPs. Current modern development methods usually do not provide tools, techniques, or guidelines on how to utilise legacy functionality (van den Heuvel et al., 2002). In (Dabous, 2005), we have identified a number of architectural patterns that address the architectural design of BPs by utilising legacy systems functionalities that correspond to some activities of BPs. Each of these patterns has consequences that are described in terms of quality attributes estimations that impact the selection of the appropriate pattern as an architectural solution for a given problem context. This papers contribution is to present and discuss two consequence models that are development and maintenance efforts validated on a real life case study in the domain of e-finance.

## 2 BASIC ASSUMPTIONS

This section presents the basic assumptions made in this paper and introduces the concepts such as functionality, legacy systems and BPs. The assumptions model used in this research is based on several architectural analysis and benchmarking studies such as (Rabhi et al., 2003) that have been conducted on a number of legacy systems.

### 2.1 Review of notation

A *functionality* in the context of this research refers to an identified autonomous task that resides within an "encapsulating entity". A functionality corresponds to an activity within a BP which performs a specific job (i.e. in part of the business logic). A functionality

T. Dabous F., A. Rabhi F., Yu H. and Al-Naeem T. (2005).
ESTIMATING PATTERNS CONSEQUENCES FOR THE ARCHITECTURAL DESIGN OF E-BUSINESS APPLICATIONS.
In *Proceedings of the Seventh International Conference on Enterprise Information Systems*, pages 248-254
DOI: 10.5220/0002553702480254

can be either automated or non-automated. If automated then the encapsulating entity can be a legacy system. On the other hand, if not automated then the encapsulating entity can be a human process. We use the notion $F^{all} = \{f_i : i = 1..|F^{all}|\}$ to represent the set of all functionalities (automated and not automated) that belong to a particular domain. The definition of the set $F^{all}$ does not tell anything about the automation of any functionality. We also use the concept of "equivalent functionalities" to refer to a group of functionalities that have similar business logic each of which resides in a different encapsulating entity. We use the notion $Q = \{q_i : i = 1..|Q|\}$ to represent the set of all groups of equivalent functionalities. Each $q_i \subseteq F^{all}$ is the $\imath^{th}$ set of a number of equivalent functionalities such that $|q_i| \geq 1$ and $q_a \cap q_b = \phi : a \neq b$ and $\bigcup_{i=1}^{q} q_i = F^{all}$ (see figure 1).

Two assumptions related to the legacy systems are made. The first one is that each legacy system is owned by one company within the domain of study and their development teams are not related to the BPs development team. The second one is that the development team for the BPs can only interact with these legacy systems through their defined interfaces (e.g. in the form of APIs) and has no access permission to the corresponding source code. Therefore, we assume that different functionalities within the same legacy system have similar interfacing mechanism.

We use the notation $F^{au} \subseteq F^{all}$ to represent the set of all automated functionalities contained in the legacy systems of a particular domain. Let $LG = \{l_i : i = 1..|LG|\}$ be the set of key legacy systems identified in that particular domain. Every $l_i \subseteq F^{au}$ and $l_a \cap l_b = \phi$ when $a \neq b$ . It is also important to note that in practice there is no instance of two equivalent functionalities within the same legacy system meaning that if $f_x \in l_i \text{and}, f_y \in l_i \text{ then } \{f_x, f_y\} \nsubseteq q \forall q \in Q$ (see figure 1).

We also consider a fixed number of BPs in a particular domain referred to by the set $BP = \{bp_i : i = 1..|BP|\}$. We assume that these BPs may have activities that correspond to existing functionalities in the legacy systems. The business logic of each $bp_i$ is expressed in terms of an activity diagram whose nodes are the activities (i.e. functionalities) and the arcs determines the execution flow between the functionalities. We use the function $activities(bp) \subseteq F^{all}$ to identify the set of functionalities that are required by the BP *bp* (i.e. it returns the set of all nodes in a *bp*'s activity diagram) and therefore the set of non-automated activities for *bp* are referred to by $\{f : f \in (F^{all} - F^{au}), f \in activities(bp)\}$. A *bp* is said to by fully-automated if $activities(bp) \subseteq F^{au}$, non-automated if $activities(bp) \subseteq (F^{all} - F^{au})$. and semi automated otherwise.

In the context of this research we assume that every $f \in F^{all}$ has at least one corresponding $bp \in BP$ such that $f \in activities(bp)$. In other words, $\bigcup_{i=1}^{b} activities(bp_i) = F^{all}$.

## 2.2 Common architectural description

We use a common architectural description for the purpose of facilitating a unified presentation the architectural patterns. This allows all pattern solutions to be expressed in a uniform way and their consequences estimated in a systematic way. An architecture is described in terms of a set $Comp = \{X_i : i = 1..|Comp|\}$ whose entries are architectural components each of which may correspond to a different software architectural entity. Components communicate with each other according to the BPs description. The features for each $X_i \in Comp$ are modelled by the following functions:

1. $tasks(X_i)$: is a function that identifies the set of tasks that are encapsulated in the component $X_i$. These tasks can be one of three types. The first one is the implementation of a functionality $f \in F^{all}$ that is denoted by $C(f)$. The second one is the implementation of a wrapper for a functionality $f$ that is denoted by $CW(f)$. It is used when $X_i$ corresponds to a 'basic service' that wraps an $f \in F^{au}$. The third one is the implemention of the business logic choreography of a BP $bp$ denoted by $CBL(bp)$. It is used when $X_i$ corresponds to a 'composite service' $x \in (BP \cup Q)$.
2. $connectTo(X_i) \subseteq Comp$: is a function that returns the set of components that $X_i$ invokes while executing its business logic.
3. $invokedBy(X_i) \subseteq Comp$: is a function that returns the set of components that invoke $X_i$.
4. $access(X_i)$: a function that returns the access method that is used by other $X_j \in Comp$ to invoke $X_i$. We use a set $AC$ of identified access methods. There are three categories of access methods that we consider: (1) service-oriented (SO) that presumes the existence of accessible interfaces by means of remote invocation using XML based protocols such as SOAP, (2) legacy-oriented that presumes the existing of APIs for local invocations whereas extra code is required to make these APIs available for remote invocations by means of binary protocols that ranges from TCP/IP to RPC based protocols, and (3) 'nil' (when $X_i$ is not required for invocation by any other $X_j \in Comp$.).

## 2.3 Proposed architectural patterns

In (Dabous, 2005), we have identified a number of architectural patterns for e-business applications using a process of combining matched design strategies. The work in this paper is based on five identified patterns briefly described as follows:

**Reuse+MinCoordinate (Pt1)**. This pattern considers accessing the required functionalities across domain legacy systems by direct invocation through the native APIs of these systems. On the other hand, each BP implements locally the activities that have no corresponding functionality in these systems.
**Reuse+Automate+MinCoordinate (Pt2)**. This pattern considers accessing the required functionalities across enterprise systems by direct invocation through their native APIs. It also considers implementing all activities of BPs that have no corresponding implementation in any of the existing systems as shared e-services with advertised service-based interfaces.
**Reuse+Wrap+Automate+MinCoordinate (Pt3)**. This pattern considers providing unified e-services to all functionalities embedded in legacy systems by developing wrappers. It also considers implementing all activities of BPs that have no corresponding implementation in any of the existing systems as shared e-services.
**Reuse+Wrap+MinCoordinate(Pt4)**. This pattern considers providing unified e-services to every functionality across all legacy systems by developing wrappers. On the other hand, each BP implements locally the activities that have no corresponding functionality in any of the existing systems.
**Migrate+MinCoordinate (Pt5)**. This pattern considers replacing existing systems with new ones. This involves migrating the implementation of required functionalities into unified e-services. To minimize redundancy, each group of equivalent functionalities is replaced with a single program code that as an e-service. For more details on the identification process of these patterns and their detailed architectural descriptions, see (Dabous, 2005).

## 3 CONSEQUENCES MODELS

Patterns consequences are typically discussed in terms of a number of quality attributes. In this paper, we only consider two quality models that estimate the patterns consequences. These qualities are development effort (devE) and maintenance effort (maintE). Each model can be applied to any of the identified patterns whose solution is formalised using the shared architectural description. The quantitative estimates generated by these models are for the purpose of comparison between the different patterns. Therefore, quality factors that are shared among all patterns and not affected by the architecture of any particular pattern are negligible and not being considered.

The two consequences models presented in this section utilise the following input values:

1. An estimate of the devE for each $f \in F^{all}$ that is denoted by the functions $FdevE(f) : f \in F^{all}$. This estimation corresponds to the effort of developing the code for a functionality as if it is built from the scratch regardless of being a functionality in a legacy system. We consider Person Months (PM) as a measurement for this estimation.
2. An estimate of the devE for each access method $ac \in AC$ that is denoted by the functions $AcDevE(ac)$. This estimation corresponds to the effort of developing a code to access a component that has an $ac$ access method. We also consider PM as a measurement for this estimation.
3. The probability for initiating a code maintenance request. There are two parts for such probability denoted by the function $modProb(X_i, C(f)) : C(f) \in tasks(X_i)$. The first one is for the code of a functionality that is in a legacy system (i.e. $X_i \in XLG$). The latter one is for the code of a new functionality (i.e. $X_i \notin XLG$). Associated with each probability is the percentage of the code to be modified denoted by the function $modPerc(X_i, C(f))$. In the first case, when $X_i \in XLG$, this percentage is equal to 100%. This is based on an assumption made in the problem context that the development team for the BPs do not have rights to access permission to the legacy code.

Methods for obtaining such values from the domain problem context are discussed in (Dabous, 2005).

### 3.1 Development effort

The overall development effort (devE) for any pattern solution encompasses the development effort associated with all tasks in all components of the resulting architecture. Considering the three types of tasks that can appear in any of the $Comp$ components, we can identify a generic development effort model as follows:

$$devE(Comp) = \sum_{X_i \in Comp} \; \sum_{t \in tasks(X_i)} XdevE(X_i, t) \quad (1)$$

In the above model, $XdevE(X_i, t)$ corresponds to the development effort for a task $t$ that is in the component $X_i$ (i.e. $t \in tasks(X_i)$). Each of the three task types is estimated differently as follows:

$XDevE(X_i, C(f))$ is nil when $C(f)$ is a task within a component that corresponds to a legacy system (i.e. $X_i \in XLG$). This is logically consequence of the fact that $f \in F^{au}$ is a functionality that is already encapsulated within a legacy system that is accessed through its advertised API. Otherwise, when $C(f)$ is a task within a component that does not correspond to a legacy system (i.e. $X_i \notin XLG$)), the $XDevE(X_i, C(f))$ is estimated by considering the value of the input function $FdevE(f)$ that uses known cost estimation models such as COCOMO or experience as discussed later in this section. Therefore:

$$XDevE(X_i, C(f)) = \begin{cases} 0 & X_i \in XLG \\ FdevE(f) & X_i \notin XLG \end{cases} \qquad (2)$$

The $XdevE(X_i, CW(f)) : f \in F^{au}$ is estimated as the effort of developing a wrapper to another component that encapsulates $f$. On the other hand, $XdevE(X_i, CBL(bp)) : bp \in BP$ is estimated only by the effort of developing accesses $\forall x \in conTo(X_i)$. We argue that the effort incurred in developing the business logic itself for $bp$ can be dropped because this effort is similar across all patterns for a given business process and this is not needed when using the estimations for comparable purposes. Therefore, we can represent both $XdevE(X_i, CW(f))$ and $XdevE(X_i, CBL(bp))$ with the same model as follows:

$$XdevE(X_i, t) \leq \sum_{X_y \in ConTo(X_i)} AcDevE(access(X_y)), t = CBL() \text{ or } CW() \qquad (3)$$

The application of (3) across all components in the architecture shows that the histogram of the usage for $AcDevE(ac)$ calls can be relatively high. Therefore, we used the operator "$\leq$" in the above model to emphasize the fact that repeating alike effort by the development team certainly enhance the learning curve and therefore the effort is decreased as the number of repetitions increases. In other words, each type of access method $ac$ across all components in $Comp$ (e.g. 'so', 'api1', 'api2', 'api3') is accessed by a number of other components in $Comp$. We assume that the different accesses of each type have similar development effort. Therefore, we have utilised the cumulative average-time learning model for this purpose. In order to apply this model, we first need to find the number of accesses (i.e. invocations) across all components in $Comp$ to the components with the similar access method. This is determined by the function $noAccesses(ac)$ that iterates through all components with 'ac' access while counting the total number of links to these components. Therefore, when applying the learning model then the average effort of developing an access to a componentt that has $ac$ access method is expressed by the functions $avgAcDevE(ac)$. This is determined as follows:

$$avgAcDevE(ac) = AcDevE(ac) * noAccesses(ac)^{log(lc)/log(2)}$$
$$\text{s.t: } lc : \text{ the learning curve (e.g. 95\%)} \qquad (4)$$

When considering the learning model, then the model in (3) is refined as follows:

$$XdevE(X_i, t) = \sum_{X_y \in ConTo(X_i)} avgAcDevE(access(X_y)), t = CBL() \text{ or } CW() \qquad (5)$$

## 3.2 Maintenance effort

The maintenance effort (maintE) that we consider is an estimation of the effort that is spent on maintaining the implemented code and deployed architecture. We emphasize the following three assumptions that have been considered in the maintenance model in this section. The first one is that the development team does not have the access rights to change or maintain the actual code for any of the legacy systems functionalities. As stated in the problem context, they can only access these functionalities through there advertised APIs. Therefore, maintaining any of these functionalities would require the development team to code it from scratch. The second one is that the maintenance of each of the new functionalities code that has been developed is maintained by directly modifying portions of the code. The last one is that the maintenance effort for the BPs logic code is insignificant because it is constant across all alternative patterns and therefore would not have an impact when comparing the estimations of maintenance across all patterns. The same is also applicable for the wrappers code because their maintenance is attached with the maintenance of the functionality code that is wrapped. Based on these assumptions, most of the maintenance effort is incurred due to the changes that are required on functionalities codes. It should be noted in our model that the expected effort of introducing new BPs corresponds to the development effort and has been discussed as a DevE prediction model in (Dabous, 2005). A model for estimating the maintE can be derived based on the following generic model that caters for the maintenance of all tasks across all components in the architecture.

$$maintE(Comp) = \sum_{X_i \in Comp} \sum_{t \in tasks(X_i)} XmaintE(X_i, t) \qquad (6)$$

Based on the last assumption above, both $XmaintE(X_i, CBL())$ and $XmaintE(X_i, CW())$ are insignificant. Therefore, the model can be estimated by the following refinement.

$$maintE(Comp) = \sum_{X_i \in Comp} \sum_{C(f) \in tasks(X_i)} XmaintE(X_i, C(f)) \qquad (7)$$

The value for $XmaintE(X_i, C(f))$ is affected by $X_i$ being in $XLG$ or not. Therefore, we start with discussing each case. In the first case, when $X_i \in XLG$, there is a probability estimated by $modProb(X_i, C(f))$ that the whole code for $f$ (i.e. $modPerc(X_i, C(f))$=1) is required to be replaced incurring a minimum effort estimated by $FdevE(f)$. The code maintenance approach that we consider is to create and add a new component $X_j$ to the architecture where $C(f) \in tasks(X_j)$ and $access(X_i)$ is the unified access method targeted by the code maintenance model (e.g. 'so' in the case study). Therefore, an extra effort denoted by the function $linksE(X_i, f)$ is added to the total $XmaintE(X_i, C(f))$ that corresponds to changing all invocations of $f$ to the new $X_j$ component. Such extra effort is not applicable when the legacy functionality has a wrapper component because the new functionality would have the same access as that wrapper and therefore replacing $C(f)$ instead of $CW(f)$ task. As a result, the overall $XmaintE(X_i, C(f)) : X_i \in XLG$ is estimated as the $modProb(X_i, C(f))$ fraction of the total of both $FDevE(f)$ and the $linksE(X_i, f)$. That is:

$$XmaintE(X_i, C(f)) = \\ modProb(X_i, C(f)) * \big(FDevE(f) + linksE(X_i, f)\big) \quad (8)$$

In the second case, when $X_i \notin XLG$, it is different from the previous case in two aspects. The first one is that a $modPerc(X_i, C(f))$ fraction of $FdevE(f)$ is modified because the source code is totally owned by the development team (recall that $modPerc(X_i, C(f)) : X_i \in XLG$ =1). Consequently, the latter aspect is that $linksE(X_i, f)$ is always nil since there is no need to create and add a new component to the architecture. That is:

$$XmaintE(X_i, C(f)) = \\ modProb(X_i, C(f)) * modPerc(X_i, C(f)) * FDevE(f) \quad (9)$$

The two cases above can be combined in the following model while considering whether $X_i$ is in $XLG$ or not when implementing it.

$$XmaintE(X_i, C(f)) = modProb(X_i, C(f)) * \\ \Big(modPerc(X_i, C(f)) * FDevE(f) + linksE(X_i, f)\Big) \quad (10)$$

Such that $linksE(X_i, f)$ is identified by the following algorithm:

```
linksE(Xi, f): C(f)∈tasks(Xi) {
 Let totEffort = 0; /* the total accumulated effort*/;
 If (Xi∉XLG) Then return totEffort;
 If (y ∈ invBy(Xi): CW(f) ∉ tasks(y)) Then
 For every y ∈ invBy(Xi) Do
   If (CW(f) ∈ tasks(y)) Then Continue
   If(f∈activities(bp): CBL(bp)∈tasks(y)) Then
      totEffort += avgAcDevE( ac);
 return totEffort;}
```

## 4 CASE STUDY

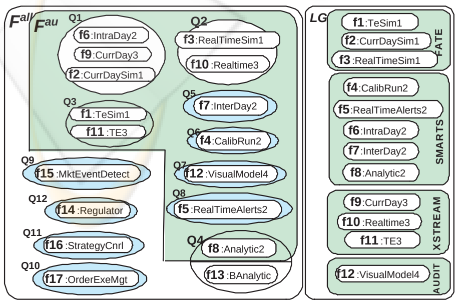

Figure 1: Case study problem context illustrated

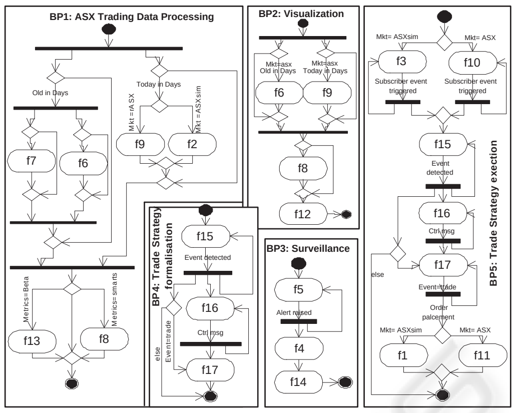

Figure 2: Activity diagrams of the BPs

### 4.1 Selected application domain

Within the e-finance domain, we focus on capital markets which are places where financial instruments such as equities, options and futures are traded (Harris, 2003). The trading cycle in capital markets comprises a a number of phases which are: pre- trade analytics, trading, post-trade analytics, settlement and registry. At each phase of this cycle, one or more legacy systems may be involved. Therefore, a vast number of BPs exist within this domain involving a number of activities that span through different stages of the trading cycle. Many of these activities can be automated by utilising functionalities of existing legacy systems. The automation of these BPs is challenging for two reasons. The first one is that it may involve a number of legacy systems that are owned by different companies. The latter one is that these BPs are normally used by business users who are not tied to any of these companies (e.g. finance researchers).

The case study presented in this paper corresponds to one problem context that comprises four legacy systems encapsulating 12 automated functionalities, five non-automated functionalities and five BPs that leverages the 17 functionalities in conducting the business logic. We focus on four legacy systems that have been customised around Australian Stock Exchange (ASX) practices. Theses systems are FATE, SMARTS, XSTREAM, and AUDIT Explorer. Each of these systems supports a number of functionalities accessible through APIs. In this paper, we consider a few functionalities in each system that are shown in figure 1. These functionalities have been reported in (Yu et al., 2004; Dabous et al., 2003). We also consider five BPs in this paper which are: ASX trading data processing, visualisation of ASX trading data, reporting surveillance alerts, trading strategy formalisation, and trading strategy execution. Figure 2 il-

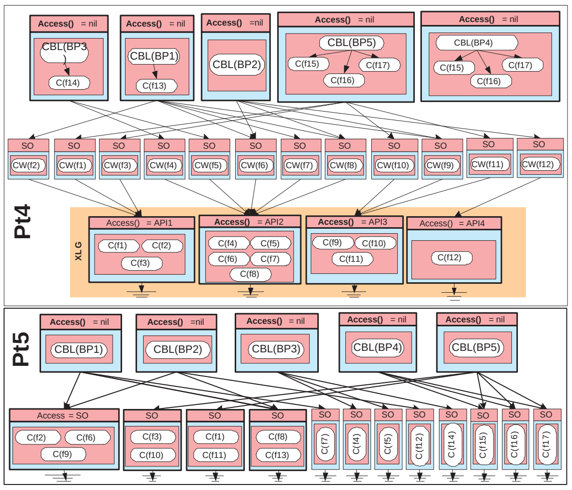

Figure 3: Pt4 and Pt5 applied on the case study

lustrates the workflow for each of these BPs as activity diagrams. More details about the functionalities, legacy systems, and the BPs that are used in this case study are discussed in (Dabous, 2005).

## 4.2 Consequences estimations

The resulting architectures are determined by applying each proposed pattern described in section 2.3 on this case study. Graphical representations are shown for Pt4 and Pt5 in figure 3. Table 1 presents the input data that is required by development and maintenance models. The columns of this table corresponds to required values as discussed in the beginning of section 3. Other predefined estimations that are required by the consequences models relate to the development effort of accessing a give $X_i \in Comp$ in particular $AcDevE(\text{access})$ (see equation (4)). We refer to the access methods used by an $X_i \in XLG$ that correspond to FATE, SMARTS, XSTREAM, and AUDIT systems as 'api1', 'api2', 'api3', and 'api4' respectively whereas 'so' access method is used as the targeted unified access method for $X_i \notin XLG$. The $AcDevE(ac)$ are 7.3, 7.3, 5.4, 3.5, 0.6 PMs for the five access methods respectively that are computed based on COCOMO model. By applying the model in l (1), the devE estimations for the five patterns are 93.58, 76.58, 117.53, 115.32, and 403.73 respectively. And by applying the model in (10), the maintE estimations for the five patterns are 48.35, 47.69, 38.50, 40.86, and 24.21 respectively.

## 4.3 Discussion

Any change to the problem context may result in different estimations for the consequences models. Such

Table 1: Case study input data

| f | FdevE(f) | modProb() | modProb() | modPerc() |
|---|---|---|---|---|
| | | $X_i \in$ XLG | $X_i \notin$XLG | $X_i \notin$XLG |
| $f_1$ | 25.4 | 10% | 10% | 50% |
| $f_2$ | 5.6 | 10% | 10% | 50% |
| $f_3$ | 7.5 | 10% | 10% | 50% |
| $f_4$ | 60.7 | 7% | 10% | 50% |
| $f_5$ | 19.3 | 7% | 10% | 50% |
| $f_6$ | 13.3 | 7% | 10% | 50% |
| $f_7$ | 4.2 | 7% | 10% | 50% |
| $f_8$ | 16.3 | 7% | 10% | 50% |
| $f_9$ | 30.5 | 12% | 10% | 50% |
| $f_{10}$ | 39.8 | 12% | 10% | 50% |
| $f_{11}$ | 134.5 | 12% | 10% | 50% |
| $f_{12}$ | 15.9 | 15% | 10% | 50% |
| $f_{13}$ | 6.3 | - | 10% | 50% |
| $f_{14}$ | 5 | - | 10% | 50% |
| $f_{15}$ | 6.3 | - | 10% | 50% |
| $f_{16}$ | 2.9 | - | 10% | 50% |
| $f_{17}$ | 3.7 | - | 10% | 50% |

changes can be changes applied to the set of functionalities, legacy systems, functionalities development effort, functionalities/legacy system modification probabilities, development effort of building interaction to an access method for a legacy system, etc. Nominating the best architectural pattern for a given problem context is not trivial when having a large set of alternative patterns together with a number of different quality models (i.e. patterns consequences) and a number of stakeholders with different preferences for qualities that are often conflicting. We have investigated a number of patterns selection methods in (Dabous, 2005). We demonstrate the application the Simple Additive Weighting (SAW) method (Hwang and Yoon, 1981) by considering the two quality models reported in this paper and the latency estimation reported in (Dabous, 2005). We only consider preferences weights provided by the development team which are 0.34, 0.33, 0.33 for the three models respectively. The application of SAW generates scaled values that are 0.65, 0.56, 0.41, 0.50, and 0.34 for the five patterns respectively. This output would suggest that the best ranking pattern is Pt1, followed by Pt2, Pt4, Pt3, and finally Pt5. The BP development team has been consulted on these results and a positive feedback has been received in supporting these results. The justification of these results is explained based on the fact that the number of identified BPs is relatively very small and therefore, Pt1 and Pt2 are most favourable. However, the development team has leveraged Pt4 then Pt3 based on the fact that more BPs are likely to be introduced and these two patterns can utilise service-oriented paradigm where high modularity and interoperability are achieved. The pattern selection method is expected to push forward towards Pt3 and Pt4 once the consequences models are extended to cope with such quality attributes.

## 5 CONCLUSIONS

E-business applications often involve a number of BPs that may require utilising legacy functionality in order to be automated. Previous work has presented a number of alternative patterns that can be utilized in such situations. In this paper, we presented two models to estimate the development and maintenance efforts as possible pattern consequences to help designers choose the appropriate pattern within a given problem context. We have validated these models by a real life case study derived from the e-finance domain. In (Dabous, 2005), we described a process of determining more patterns based on practices. As the number of patterns increases with the presence of multiple stakeholders with different and often conflicting preferences of qualities, the problem of determining the appropriate pattern to adapt becomes more difficult. In (Al-Naeem et al., 2005), we have leveraged Multiple-Attribute Decision Making (MADM) methods for this purpose. A simple tool has been developed to systematically generate estimates for the quality models and to rank patterns with accordance to their appropriateness for a given problem context. Our current research direction is to address consequences models for a number of other quality attributes. We are also investigating extending the tool support to systematically generate the architectures for all the patterns and estimations for other quality attributes.

## REFERENCES

Al-Naeem, T., Dabous, F. T., Rabhi, F. A., and Benatallah, B. (2005). Quantitative evaluation of enterprise integration patterns. In *7th Int. Conf. on Enterprise Information Systems (ICEIS05)*, USA.

Dabous, F. T. (2005). *Pattern-Based Approach for the Architectural Design of e-Business Applications*. Phd thesis, School of Information Systems, Technology and Management, The University of New South Wales, Australia. (to be submitted in Apr 2005).

Dabous, F. T., Rabhi, F. A., and Yu, H. (2003). Performance issues in integrating a capital market surveillance system. In *Proceedings of the 4th International Conference on Web Information Systems engineering (WISE03)*, Rome, Italy.

Harris, L. (2003). *Trading and Exchanges: Market Microstructure for Practitioners*. Oxford University Press.

Hwang, L. and Yoon, K. (1981). Multiple criteria decision making. *Lec. Notes in Economics and Mathematical Systms*.

Rabhi, F. A., Dabous, F. T., Chu, R. Y., and Tan, G. E. (2003). SMARTS benchmarking, prototyping & performance prediction. Technical Report CRCPA5005, Capital Market Cooperative Research Center (CMCRC).

Umar, A. (1997). *Application (Re)Engineering: Building Web-Based Applications and Dealing With Legacy systems*. Prentice Hall.

van den Heuvel, W.-J., van Hillegersberg, J., and Papazoglou, M. (2002). A methodology to support web-services development using legacy systems. In *IFIP TC8 / WG8.1 Working conference on Engineering Information Systems in the Internet Context*.

Yu, H., Rabhi, F. A., and Dabous, F. T. (2004). An exchange service for financial markets. In *6th Int. Conf. on Enterprise Information Systems (ICEIS04)*, Porto, Portugal.