A FRAMEWORK FOR IDENTIFYING ARCHITECTURAL PATTERNS FOR E-BUSINESS APPLICATIONS

Feras T. Dabous¹, Fethi A. Rabhi¹ and Tariq Al-Naeem²

¹School of Information Systems, Technology and Management
²School of Computer Science and Engineering
The University of New South Wales, 2052 Sydney NSW, Australia


Abstract: The success of today’s enterprises is critically dependant on their ability to automate the way they conduct business with customers and other enterprises by means of e-business applications. Legacy systems are valuable assets that must play an important role in this process. Selecting the most appropriate architectural design for an e-business application would have critical impact on participating enterprises. This paper discusses an initiative towards a systematic framework that would assist in identifying a range of possible alternative architectural designs for such applications and how some of these alternatives can evolve into formal architectural patterns or anti-patterns. This paper focuses on e-business applications with some requirements and assumptions that are often presented in a few specific domains such as e-finance. The concepts presented in this paper are demonstrated using a real life case study in the domain of e-finance and in particular capital markets trading.

1 INTRODUCTION

The concept of e-business applications has been used extensively in the literature to refer to a range of applications. This range encompasses B2C interactions (e.g. simple Web-based client/server applications) and B2B interactions either as a workflow by means of workflow technology or as a distributed application by means of middleware technologies. In this paper, we consider an e-business application domain that can be improved by developing or utilising corresponding business logic and functionality that can span different organisational legacy systems.

In previous work, we identified and formalised a number of architectural patterns based on practical experience in the domain of e-finance. These patterns address a similar problem context and therefore constitute alternative architectures where the selection of the most appropriate pattern is based on the problem context specification. In (Dabous et al., 2005), we presented models for patterns qualities estimations that can be used in the selection process for a given problem context, whereas in (Al-Naeem et al., 2005b) we utilised AHP (Anderson et al., 2002) method for this selection process that is extended in (Dabous, 2005) to encompass estimations provided by the quality models.

This paper extends upon the previous work by proposing a framework for identifying patterns that can be used in the architectural design of e-business applications. We consider e-business domains that have special requirements and assumptions mainly related to the existence of legacy functionality. These domains, such as e-finance, consist of applications that are business process intensive and therefore we alternatively use the term Business Process (BP) to refer to an e-business application.

This paper is organised as follows. Section 2 presents the basic assumptions made in this research and formalises the concepts used. It also presents e-finance particularly capital markets as a real-life application domain. Section 3 discusses a common architectural description that forms the basis of the patterns identification framework. Section 4 discusses the two major phases in the proposed patterns identification framework. Section 5 presents and discusses five design strategies that ingrain the basic input for the framework. Section 6 presents five candidate patterns that are the outcome of the first phase in the proposed framework together with the instantiation of each pattern on the selected application domain. Finally, section 7 concludes this paper.
2 BASIC ASSUMPTIONS AND CONCEPTS

This section presents the basic assumptions made in this paper and introduces the different concepts used such as functionality, legacy systems and BPs together with a selected application domain. The assumptions and concepts discussed are based on several architectural analysis, benchmarking studies, and e-business applications development 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 can be either automated or new (i.e. non-automated). If it is automated then the encapsulating entity can be a legacy system. On the other hand, if it is new functionality then the encapsulating entity can be a human process. We use the notion $F^\text{all} = \{f_i : 1 \leq i \leq |F^\text{all}|\}$ to represent the set of all functionalities (automated and not automated) that belong to a particular domain. The definition of the set $F^\text{all}$ 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 $Q = \{q_i : 1 \leq i \leq |Q|\}$ to represent the set of all groups of equivalent functionalities. Each $q_i \subseteq F^\text{all}$ is the $i$th set of a number of equivalent functionalities such that $|q_i| \geq 1$ and $q_a \cap q_b = \emptyset : a \neq b$ and $\bigcup_{i=1}^{|Q|} q_i = F^\text{all}$.

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 the 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 $F^\text{auto} \subseteq F^\text{all}$ to represent the set of all automated functionalities contained in the legacy systems of a particular domain. Let $LG = \{l_i : 1 \leq i \leq |LG|\}$ be the set of key legacy systems identified in that particular domain. Every $l_i \subseteq F^\text{auto}$ and $l_a \cap l_b = \emptyset$ 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 $f_a \in l_i$ and $f_b \in l_i$, then $\{f_a, f_b\} \not\subseteq q_i \forall q_i \in Q$.

We also consider a fixed number of BPs in a particular domain referred to by the set $BP = \{bp_i : 1 \leq i \leq |BP|\}$. 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 determine the execution flow between the functionalities. We use the function $activities(bp) \subseteq F^\text{all}$ to identify the set of functionalities that are required by the BP $bp_i$ (i.e. it returns the set of all nodes in a $bp_i$’s activity diagram).

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

2.2 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 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 application domain 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
Figure 1: Selected application domain

Figure 2: Activity diagrams of the BPs

2 illustrates 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 application domain are discussed in (Dabous, 2005).

3 COMMON ARCHITECTURAL DESCRIPTION

We propose a common architectural description with the main motivation of facilitating a unified systematic architectural presentation. It describes each identified pattern in terms of a set of architectural components \( \text{Comp} = \{ X_i : 1 \leq i \leq |\text{Comp|} \} \) which communicate with each other according to the BP description. A component \( X_i \in \text{Comp} \) is described according to four essential features: denoted by the tuple \(< \text{tasks}(X_i), \text{conTo}(X_i), \text{invBy}(X_i), \text{access}(X_i) > \) where:

1. \text{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 \( f \in F^{al} \) that is denoted by \( C(f) \) and is used in three different cases. The first is when \( f \in F^{al} \) refers to an existing legacy functionality within a legacy system. The second is when \( f \in F^{au} \) refers to redeveloping (i.e. reengineering) an existing functionality. The third is when \( f \in (F^{al} - F^{au}) \) which is not implemented in any of the legacy systems. The second one is the implementation of a wrapper for a functionality \( f \) that is denoted by \( CW(f) \). It is used when \( X_i \) makes it possible to invoke the functionality \( f \) from other components. The third one is the implementing of the business logic of a business process \( bp \) denoted by \( CBL(bp) \). We will use the term ‘business process enactment’ to refer to the implementation code of the business logic. Note that these three tasks functions return the implementation code resulted from applying the respective task on its parameters. For example \( C(f) \) returns the implementation code when applying the ‘create’ task on \( f \) (i.e. the code that implements \( f \)).

2. \text{conTo}(X_i) \subset \text{Comp}: is a function that returns the set of components that \( X_i \) invokes while executing its business logic.

3. \text{invBy}(X_i) \subset \text{Comp}: is a function that returns the set of components that invoke \( X_i \).

4. \text{access}(X_i): a function that returns the access method that is used by all \( X_j \in \text{invBy}(X_i) \) to invoke \( X_i \). There are three main categories of access methods that we consider. The first one is service-oriented (SO) that presumes the existence of accessible interfaces by means of a remote invocation using XML based protocols such as SOAP. The second one is legacy-oriented that presumes the existence 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. For this access method we used a function denoted by \( api(l) : l \in LG \) that returns the API name of the legacy system \( l \) that we also use as the access method for the same legacy system. The third one is ‘nil’ that is used when \( X_i \) is not required for invocation by any other \( X_j \in \text{Comp} \). We denote the set of all available access methods by the set \( AC \) and therefore \( \text{access}(X_i) \in AC \forall X_i \in \text{Comp} \).
design strategies. This phase consists of two main steps: structural descriptions that correspond to candidate patterns. This stage considers the identification of architectural patterns for e-business applications.

This section presents an extensible systematic framework for identifying domain-specific architectural patterns for e-business applications. As illustrated in figure 3, this framework comprises two phases: candidate patterns identification phase followed by the evolution phase into formal patterns.

**Phase #1: candidate patterns identification**

This stage considers the identification of architectural descriptions that correspond to candidate patterns. This phase consists of two main steps:

1. Given a set of design strategies denoted by the DS (see section 5) such that each strategy is formalised using the common architectural description that is discussed in section 3, generate the power set of all subsets denoted by \( P(DS) \) that contains \( 2^{|DS|} \) elements. Each \( c \in P(DS) \) corresponds to a unique possible combination of design strategies.

2. For each \( c \in P(DS) \) do the following: Firstly, let \( arch_c \) be the combined architectural construction algorithms \( \forall d \in c \). Secondly, having a set of combining rules denoted by Rules (see section 5.2), check \( arch_c \) against all combining rules. If \( arch_c \) is validated \( \forall r \in \text{Rules} \), then \( arch_c \) is added to the set of candidate patterns denoted by \( P'^{\text{cndt}} \) (i.e., \( P'^{\text{cndt}} = P'^{\text{cndt}} \cup \{arch_c\} \)), otherwise discard \( arch_c \).

**Phase #2: evolution into formal patterns**

This stage considers the evolution opportunities for the candidate patterns in \( P'^{\text{cndt}} \) into either formal patterns denoted by the set \( P'^{\text{frml}} \) or anti-patterns denoted by the set \( P'^{\text{anti}} \). Theoretically, this evolution process would result in \( P'^{\text{frml}} = P'^{\text{frml}} \cup P'^{\text{anti}} \).

However, in reality, the evolutionary process for any candidate pattern may take several years and depends mostly on frequent use for that candidate pattern in practice with different problems to generate either successful solutions that push forward into formal pattern or unsuccessful solutions that push forward into anti-pattern. This phase consists of the five main steps:

1. Instantiate the architectural description for each identified candidate pattern in \( P'^{\text{cndt}} \) on a given application domain problem that is formalised with accordance to the notations in section 2.1. Section 6 discusses the candidate patterns instantiations on the application domain that is illustrated in section 2.2.

2. Apply the consequences estimation models, which are discussed in (Dabous et al., 2005; Dabous, 2005), on the given application domain specifications so that estimation per model for each pattern architecture is generated.

3. Investigate the stakeholders’ preferences on the required quality attributes that correspond to the patterns’ consequences.

4. Apply a pattern selection algorithm based on steps 1, 2, and 3 using common MADM methods such AHP which is discussed in (Al-Naeem et al., 2005). The outcome of such a selection method is a rank for each pattern architecture in \( P'^{\text{frml}} \).

5. The last step is to investigate possibilities of the evolution of each candidate pattern. A candidate pattern would gradually evolve into a formal pattern in \( P'^{\text{frml}} \) if it has been occupying one of the higher ranks and successful stories have been reported about its usage in practice. Otherwise, such candidate pattern would gradually evolve into an anti-pattern in \( P'^{\text{anti}} \) if it has been occupying lower ranks and/or unsuccessful stories have been reported about its usage in practice. While evolving, each candidate pattern documentation in \( P'^{\text{frml}} \) would also be enriched with all experiences with different problems context together with the ranks, successful and unsuccessful stories.

Therefore, this framework is extensible because the incorporation of one extra design strategy generates
extra $|DS|$ combinations, and therefore possibly motivates extra candidate patterns wherever a valid combination is generated. The framework is also systematic since all design strategies, candidate patterns, and consequences models are defined around one common architectural description (the section 5), and therefore a tool can be developed in order to navigate through the steps of this framework. Due to space limitations, this paper extends only to the first phase of demonstrating the identification of a few candidate patterns and step 1 in the second phase by applying a realistic application from the domain of e-finance. Indeed, related work to step 2 is reported in (Dabous et al., 2005) and related work to steps 3 and 4 is reported in (Al-Naeem et al., 2005b; Al-Naeem et al., 2005a). Step 5 of phase 2, which requires long term validation, is not yet addressed thoroughly in this research.

5 DESIGN STRATEGIES AND COMBINATION RULES

In this framework, we use the concept of ‘design strategy’ as a basic architectural construct into the pattern identification process. Each design strategy is described using the common architectural description. We use the notion $DS = \{\text{Reuse, Automate, Wrap, Migrate, MinCoordinate}\}$ as the set of design strategies used in this paper. We denote a set $XLG$ to represent the architecture by creating a component for every legacy system. This set of components is presumed to exist. This set is constructed using the following algorithm:

**Construct** $XLG (LG)$
- For each $l_x \in LG$ Do:
  1. $X_x = \text{createNew}(); /* creates and initialises a new component*/
  2. access$(X_x) = \text{api}(l_x)$;
  3. tasks$(X_x) = \{C(f_k) : f_k \in l_x\}$;
  4. $XLG = XLG \cup \{X_x\}$ /* add $X_x$ to $XLG$*/

Where createNew() function would create a component $X_x$ of the same type as the components of $Comp$ and initialise its four features to nil (i.e. access()= tasks()= conTo()= invBy()= nil). Whereas api($l_x$) functions returns the access method that is supported by the legacy system $l_x$.

5.1 Design strategies

**Reuse** This design strategy considers the existing legacy systems and the functionalities they embed as existing assets that should be utilised immediately. Based on the proposed generic architecture, this design strategy corresponds to creating $|LG|$ entries in $Comp$ which have been already declared as the content of the set $XLG \in Comp$. The non-automated functionalities (i.e. $\forall f \in (F_{all} - F_{au})$) are not addressed in this design strategy. Its corresponding algorithm is:

\* **Reuse()**

- For each $X_i \in XLG$:
  - $Comp = Comp \cup \{X_i\}$;

**Automate** This design strategy creates $|F_{all} - F_{au}|$ components each of which supports a new functionality that does not exist in any of existing legacy systems. This strategy implements a unified access type for these new functionalities (i.e. $\forall f \in (F_{all} - F_{au})$). This is accomplished using the following algorithm:

\* **Automate(ac) */ac is the access method used*/

- For each $f_k \in (F_{all} - F_{au})$:
  1. $X_x = \text{createNew}();$
  2. access$(X_x) = ac$;
  3. tasks$(X_x) = \{C(f_k)\}$;
  4. $Comp = Comp \cup \{X_x\}$;

**Wrap** This design strategy considers a unified access method for all functionalities identified in the set of legacy systems (i.e. $\forall f_i \in F_{au}$). It creates a total of $|F_{au}|$ components in $Comp$ each of which corresponds to a wrapper component (i.e. for every functionality that resides in a legacy system). This is accomplished using the following algorithm:

\* **Wrap (ac)** /*ac is the access method used*/

- For each $X_i \in XLG$ such that $X_i \in Comp$ do:
  - For each $C(f_k) \in \text{tasks}(X_i)$:
    1. $X_x = \text{createNew}();$
    2. access$(X_x) = ac$;
    3. tasks$(X_x) = \{C(f_k)\}$;
    4. invBy$(X_i) = X_x$
    5. conTo$(X_i) = X_x$
    6. $Comp = Comp \cup \{X_x\}$

**Migrate** This design strategy ignores existing legacy systems and therefore redevelops the functionalities $\forall f \in F_{all}$ with a unified access method. This strategy groups each set of equivalent functionalities $q \in Q$ in one component. Therefore, it creates a total of $|Q|$ new components in $Comp$ where each new component corresponds to one $q \in Q$ and contains a task $C(f_k)$ for each $f_k \in q$. This is accomplished using the following algorithm:

\* **Migrate(ac) */ac is the access method used*/

- For each $q_i \in Q$:
  1. $X_x = \text{createNew}();$
  2. access$(X_x) = ac$;
  3. tasks$(X_x) = \{C(f_k) : f_k \in q_i\}$;
  4. $Comp = Comp \cup \{X_x\}$
MinCoordinate Minimum Coordinate (MinCoordinate) design strategy relates to the implementation of every bpj ∈ BP enactment as an autonomous component that depends directly on the activities denoted by activities(bpj). This design strategy mandates that any bpj ∈ BP should implement its own private copy of any required functionality that is not supported in any other component in Comp. Meaning that an f ∈ Fnull such that C(f) ∉ tasks(Xj) ∀ Xj ∈ Comp should implement an instance of f that is accessed only by Xj and therefore tasks(Xj) = tasks(Xj) ∪ {C(f)}. This design strategy is accomplished using the following algorithm:

MinCoordinate()
1. Let BpComp = φ/* An empty set of same type as Comp/*
2. For each bpj ∈ BP:
   (a) Xs = createNew();
   (b) access(Xs) = φ;
   (c) tasks(Xs) = {CBL(bpj)};
   (d) For each f_k ∈ activities(bpj) DO:
       • IF (∃Xj : C(f_k) ∈ tasks(Xj))
         - conTo(Xs) = conTo(Xs) ∪ {Xj};
         - invBy(Xs) = invBy(Xs) ∪ {Xj};
       • ELSE tasks(Xs) = tasks(Xs) ∪ {C(f_k)};
   (e) BpComp = BpComp ∪ {Xs};
   3. Comp = Comp ∪ BpComp;

5.2 Design strategies combination rules

The design strategies are combined by linearly applying their corresponding construction algorithms. The order in which the participating design strategies are combined with the aim of generating a ‘valid’ combination is significant. This order is based on associated rank for each strategy. The first rank is for ‘Reuse and ‘Migrate’, the second is for ‘Automate’ and ‘Wrap’, the third is for ‘MinCoordinate’ design strategy. The naming convention for the patterns that we consider shows the participating design strategies in the correct combining order.

We have investigated the major combining rules that are applied to any possible design strategies combination and therefore considering this combination as either valid or invalid. The main idea for such rules is to make sure that the combined design strategies algorithms can form a neat architecture when applied on any problem context specification. A neat architecture would guarantee that Comp architecture facilitates the connectivity from any business process to all of its required functionalities without redundancies. The following rules are the major ones that should apply in order to consider a valid design strategies combination and therefore a candidate pattern:

1. Existence of tasks that implements all BPs logic:
   For every bp ∈ BP, ∃X : CBL(bp) ∈ tasks(X).
2. Existence of tasks that implement all functionalities in Fnull:
   For every f ∈ Fnull, ∃X : C(f) ∈ tasks(X).
3. A functionality can have one and only one wrapper component:
   If CW(f) ∈ tasks(Xi) and CW(f) ∈ tasks(Xj) such that access(Xi) = access(Xj) then Xi = Xj.
4. Non-redundancy of a BP logic implementation task:
   If CBL(bp) ∈ tasks(Xi) and CBL(bp) ∈ tasks(Xj) then Xi = Xj.
5. Redundancy of a functionality implementation can be found locally in any BP that requires it if that functionalities is not created publicly elsewhere:
   If C(f) ∈ tasks(Xi) and C(f) ∈ tasks(Xj) : Xi ≠ Xj then CBL(bp_a) ∈ X_i and CBL(bp_b) ∈ X_j : a ≠ b.

Otherwise, the generated combination will be ‘invalid’ and therefore it is not considered as a candidate pattern. For instance, such invalid combinations may generate the following cases:

I The existence of redundancy across the tasks of the matched design strategies (e.g. ∃C(f) : C(f) ∈ tasks(Xi) and ∈ tasks(X_j) where Xi and Xj are in different design strategies).
II The existence of missing tasks that one of the participant design strategies should contain (e.g. For any f ∈ Fnull ∃Xi : C(f) ∈ tasks(Xi)).

By applying these rules, we obtained seven valid combinations each of which constitutes candidate pattern. The first five patterns are more likely to evolve into formal patterns because they are initially identified based on practical experience and reported in (Dabous, 2005) whereas the other two are expected to evolve into anti-patterns because they promote the isolation of standalone BP implementation that is obsolete and does not correspond to current practices.

6 CANDIDATE PATTERNS

Now we briefly discuss the architectural description of the five candidate patterns. Details of their forces and consequences are discussed thoroughly in (Dabous, 2005). We also show the architecture generated when instantiating each candidate pattern on the selected application domain (i.e. the contents of the set Comp). Figures 4-8 illustrate visually the contents of the set Comp for these five architectures. In these figures, each box (i.e. component) corresponds to an Xi ∈ Comp showing its tasks as ovals and access method on the top of the box whereas each arrow (i.e. link) from Xa to Xb means that Xa ∈ conTo(Xb) and Xa ∈ invBy(Xb). The patterns instantiations of
the application domain correspond to the second step of the second phase in the proposed framework. We provide our naming convention for each pattern as follows:

**Reuse+MinCoordinate (Pt1)** This pattern considers generating invocations to the required functionalities across legacy system by direct invocations through the APIs of these systems. On the other hand, each BP implements its local program code for each of its activities that have no corresponding functionality in any of the legacy systems. Figure 4 shows the architecture of Pt1 instantiation on the selected application domain.

**Reuse+Automate+MinCoordinate (Pt2)** This pattern considers generating an invocation to the required functionalities across legacy system by direct invocation through the APIs of these systems. It also considers implementing all activities of BPs that have no correspondence in any of the legacy systems as global e-services with advertised service-based interface. Figure 5 shows the architecture of Pt2 instantiation on the selected application domain.

**Reuse+Wrap+Automate+MinCoordinate (Pt3)** This pattern considers providing a unified service-based interface to every particular functionality across legacy systems. This provision is achieved by developing service-based Wraps. It also considers implementing all activities of BPs that have no correspondence in any of the legacy systems as global e-services with advertised service-based interface. Figure 6 shows the architecture of Pt3 instantiation on the selected application domain.

**Reuse+Wrap+MinCoordinate (Pt4)** This pattern considers providing a unified service-based interface to every particular required functionality across legacy systems. This provision is achieved by developing service-based Wraps. On the other hand, each BP implements its local program code for each of its activities that have no corresponding functionality in any of the legacy systems. Figure 7 shows the architecture of Pt4 instantiation on the selected application domain.

**Migrate+MinCoordinate (Pt5)** This pattern considers the disposal or abandoning existing legacy systems. Therefore, starting from scratch, this pattern migrates the implementation of required functionalities through a re-engineering/automation process into global e-services with advertised service-based interface. In this process, each group of equivalent functionalities are reengineered within a single program that supports one interface.
Figure 8 shows the architecture of Pt5 instantiation on the selected application domain.

MinCoordinate (Pt6) and Automate+MinCo. (Pt7) are more likely to evolve into anti-patterns because they promote the isolation and standalone BP implementation that is obsolete and does not correspond to current practices.

7 CONCLUSION

In this paper, we presented and discussed the basic components of a systematic extensible framework for identifying patterns for the architectural design of a category of applications in the e-business domain. This work is motivated by three factors. The first one is the availability of five architectural patterns that are identified and formalised in (Dabous, 2005) based on practical experience in developing e-business applications that utilise legacy functionality in the domain of e-finance. The second one is the quality models reported in (Dabous, 2005; Dabous et al., 2005) that estimate the patterns consequences for a given application domain specification. The third one is the architectural patterns selection method based on AHP that is reported in (Al-Naeem et al., 2005b; Al-Naeem et al., 2005a).

The proposed framework is systematic and extensible. It is systematic since the formalisation of the design strategies, the candidate patterns and the quality models uses the same common architectural description. It is also extensible since the identification of each extra design strategy would duplicate the number of strategies combinations and therefore possibly triggers extra candidate patterns. In (Dabous, 2005), we illustrated the impact of introducing additional two strategies on identifying more candidate patterns.

Current work focuses on providing tool support for automatic generation of the candidate patterns architectures for a given application domain specification and for ranking the architectures generated by instantiating these patterns with accordance to their appropriateness based on both the estimations provided by the quality models and the preferences for such quality provided by stakeholders.

REFERENCES


