Extracting Business Processes from Three-Tier Architecture Systems

Maokeng Hung[1] and Ying Zou[2]
Department of Electrical and Computer Engineering
Queen's University
Kingston, ON, K7L 3N6, Canada
alex.hung@ece.queensu.ca[1], ying.zou@queensu.ca[2]

Abstract

To minimize the overall expense and to reduce time to market, organizations either modify existing source code to meet the new requirements, or reuse existing components in new systems. Unfortunately, many software systems never have up-to-date documentation. Absence of good documentation increases the challenges for maintenance because the developers must read through the source code to understand the behaviors of the systems and to locate business logics manually. In this paper, we proposed an automatic method to generate the business processes for the three-tier architecture systems by identifying the business data and business policies in the source code.

1. Introduction

Software maintenance has become one of the most critical and longest stages in the life cycle of software systems. When the documentation of a system is lost, out-dated or unavailable, it is essential to extract important information, such as architectures, designs or requirements before maintenance or reuse can take place. Often, the only reliable source for such information is either in the mind of the developers or deeply buried in the source code. It is labor-intensive to manually scan the source code to extract the documentation.

A business logic is "a requirement on the *condition* or manipulation of data expressed in terms of the business enterprise or application domain" [1] and a business policy specifies the rules and conditions on when and where the business logic should be executed [2]. The business logics and policies form the business processes which organizations specify how they run their business. For instance, when a book is ordered from an online store, the business process consists of checking the availability of the books, restocking the inventory if the book is out-of-stock, validating the credit card and shopping to the customer; each of the individual tasks is a business logic and the condition to execute a specific task is the business policy (i.e. "book is out-of-stock" is the condition for "restocking the inventory"). It is, however, the nature of the business process to change fast in order to adapt to the market dynamics [3]. As a result, automatic or semi-automatic methods for business process extraction are essential for organizations to remain competitive by increasing the response rate to customers' demands and by reducing the costs of reengineering tasks.

For e-commerce systems, the three-tier architecture has many advantages over the one-tier and two-tier architectures in various ways. The separation of layers between user interface, business logic and database not only distinguish the functionalities between components but also provide additional information including the business data and the explicitly defined interfaces that used by business logics to communicate to the database. In this research, we propose a technique to automatically extract the business process by analyzing the communication channel and the information flow between the business layer and the database layer from the source code of the three-tier architecture systems. Based on the database operations and the information flow of the business data, we are able to identify business policies, logics and processes from the source code.

The paper is organized as follows. We discuss the benefits of the three-tier architecture and the relationship between the three-tier architecture and business logic extraction in Section 2. Section 3 discusses the method we proposed to extract business logics by business data and policies. Our approach identifies the business data and policies from the database operations explicitly invoked in the source code. We show a case study of our approach and the improvements over our previous work from [4] in Section 4. Finally, Section 5 gives the conclusion of this paper.

2. Software Architecture and Business Logic Extraction

Currently, most e-commerce applications adapt three-tier architecture. The three-tier architecture has higher maintainability than the traditional one-tier or two-tier architectures because the components are well separated and the interface between components is well-defined.

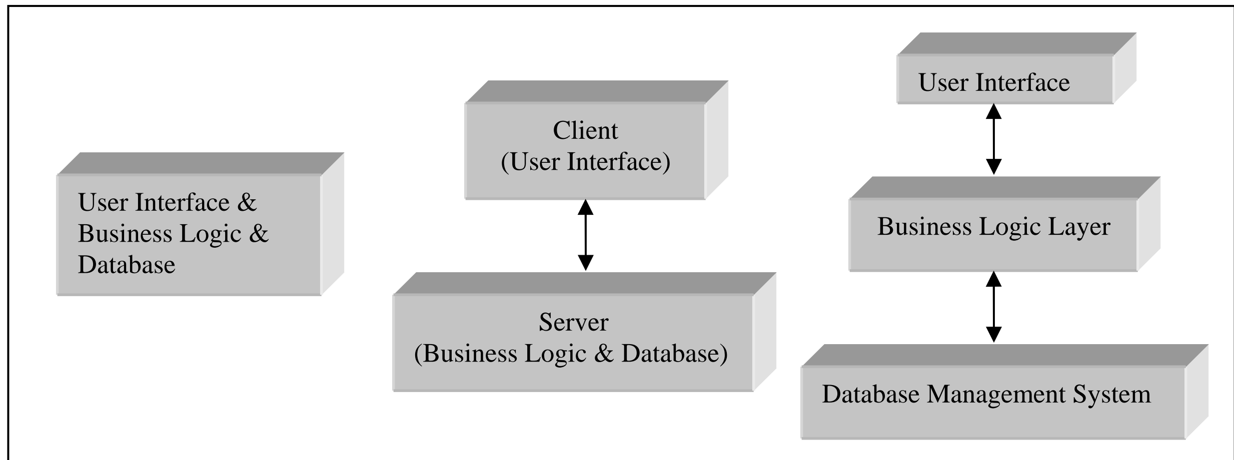

Figure 1 – One-Tier, Two-Tier and Three-Tier Software Architectures

In the three-tier architecture, user interface, business logic and database management are differentiated, as opposed to the mixture of functionalities in the one and two-tier architectures (Figure 1). Three-tier architecture provides numerous advantages over one-tier and two-tier architectures for reverse engineering and design recovery. In one and two-tier architectures, the source code entities implementing different components are interweaved with each other. It is therefore very difficult to separate the business logic components from others because of the absence of the clear partition. In three-tier architecture, the business logic and business policy components are implemented in a separate layer (the middle layer) and communicate with the user interface (UI) and the database management systems (DBMS) via external APIs. This explicit division between different functionalities not only partitions business data, policy and logics code in the source code but also clearly points out the communication of the business data and the corresponding fetch/update operations (such as J2EE EJB and embedded SQL) to the DBMS.

3. Business Process Extraction

To extract business logics from the source code, it is important to identify the relevant business entities that constitute business logics. The business entities include the business data and the business policies where the data are the inputs/outputs affecting the dataflow of the business logics and the policies control the execution path and control flow of the business logics. We employ forward and backward tracing to identify the exact location of the business logics. After the business logics are identified, the business process that determines the communications between the business logics and policies can be generated by static tracing as proposed in our previous work in [4].

3.1. Business Data

Organizations carry out their decisions based on the business policies, logics and data (input and output of the business logics). Often the input data of the business logics are fetched and the output data are updated to the DBMS. The explicit database fetch and update operations, such as the getters and setters of the J2EE® EJB objects respectively, identify the input and output of the business logics and therefore signal the presences of the business logics. Once the locations of the inputs and outputs are recognized from the source code, the static tracing technique is applied to identify where and how these data are used.

Fetch operations are often used to retrieve input data of the business logics from database. Centered on the business data, the execution of the business logics are dependent on the values of the data to make decisions and to compute the outputs of the logics. Therefore we apply the forward tracing, which analyzes the dataflow toward the direction of the execution, to the input data until the end of their lifetime. The usages of the business data of the fetch operations are discussed in the following:

a) Business policy conditions (also see Section 3.2) - Most of the decisions are made according to the information and data fetched from the database. For instance, the percentage of tax is charged differently to an online purchase according to the destination shipping address. As a result, the business data can indicate the business policy conditions and direct the execution of the business logics, as shown in Figure 2.a.
b) Computations of the output values of business logics - The output values are often computed from the business data. For instance, the final price for a purchase is calculated by the original price plus the tax. The calculated value is either used as

input of other business logics (i.e. to calculate the total) or updated to the database (i.e. to save the purchase history of a customer). Assignment statements with the business data on the right hand side is considered as an example of this computation, as shown in Figure 2.b.

c) Inputs of the user-defined functions - Method invocations are user-defined functions that perform specific tasks. Similar to b), a method can return a derived value based on the input data fetched from database; therefore, the method can be treated as one business logic where the business data are inputs and return value is the output of the logic, as shown in Figure 2.c.

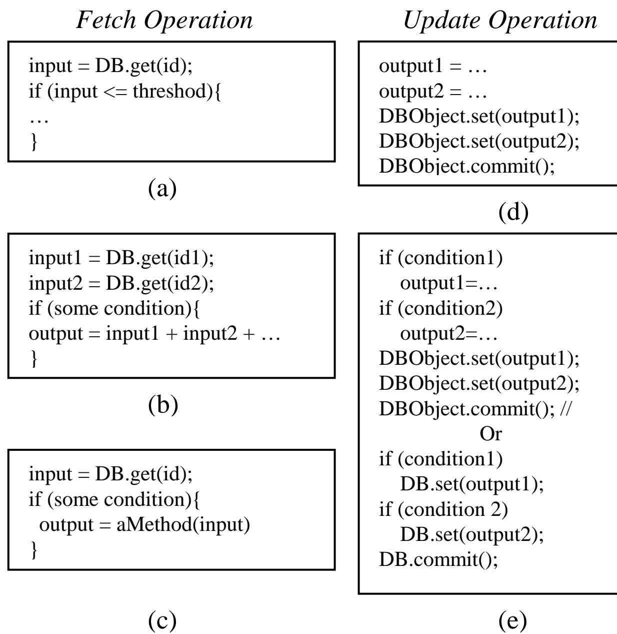

Figure 2 – Database Fetch and Update Operations and the Corresponding Business Logics

On the other hand, update operations store the result outputs of the business logics to the database. Therefore, we apply the backward tracing, which analyzes the dataflow toward reverse direction of the execution, to the parameters of the update operations. The backward tracing is able to identify the conditions and the locations where the outputs are computed or assigned. The usages of the business data of the update operations are discussed in the followings:

d) Computations performed under the same condition - If all of the parameters of the update operations are computed under the same condition (i.e. in sequence), the multiple values should be treated as a single composite data because the sequential derivations imply the grouping of the data that are updated to the database. As a result, the completion of the update operation (i.e. transaction committed to database) is the location of the business logic, as shown in Figure 2.d.

e) Computations performed under the different conditions - If any of the parameters of the update operations are computed under the different conditions, it implies that each value may or may not be present in the final transaction. Therefore, we can apply backward tracing to parameters of the update operations and identify where the computation is performed. We consider the individual computation as a business logic, as shown in Figure 2.e.

3.2. Business Policies

Business data are the center of the data flow of the business logics, and business policies determine the control flow of the business process and the execution of the business logics. In other words, business logics will be executed only when the current system states meet the conditions of the business policies. As a result, business policies signal the presences of the business logics.

However, not all conditions that guide the execution sequence of the business process are business policies. Many of the conditions are only specific to the programming languages. For instance, a program may check for current availability of the external services and it decides to wait (i.e. stay at the current state instead of moving to the next one if we treat the business process as a finite state machine). A program may also check the initialization of a variable and assign the variable with a default value if it is not initialized to avoid errors. Such conditions are only specific to program domains and have no real meaning in the business domains; thus they cannot be business policies. In our research, we define the business policies as the followings:

a) Business Policies specify the *constraints* that affect the behaviors, i.e. in an online shop, whether the purchased items can be downloaded, such as e-books or software, or it must be physically shipped, such as real books or computers.

b) Business Policies specify the *derivation of conditions that affect the execution flow*, i.e. the destination region of a shipping item according to the address and postcode.

c) Business Policies specify the *conditions* under which the computation is performed, i.e. the amount of tax to be charged to the item according to the regions.

According the above definitions, we can identify and extract business policies from the following scenarios:

1. Inside different branches of a CHOICE, an object invokes different methods or same method with different parameters, as shown in Figure 3.a. The conditions of the CHOICE are business policies.
2. Inside different branches of a CHOICE, a variable or variables are computed from different values, as shown in Figure 3.b. The conditions of the CHOICE are business policies.
3. A condition of a CHOICE is derived from business data in advance, as shown in Figure 3.c. Such condition is a business policy.

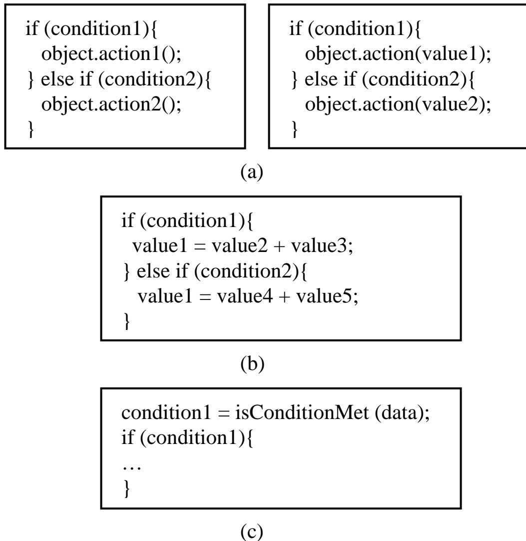

Figure 3 – Three cases of Business Policies

After business policies are identified from the source code, each execution sequence under the different conditions is grouped together as one business logic with the exception that other business logics are identified inside the execution sequence.

Business data and business policies have overlaps over each other in many occasions where the former handles the dataflow aspect and the latter handles the control flow aspect. Often, policy conditions are derived from business data. For instance, the shipping cost might be calculated based on the destination whose value, such as postcode, is fetched from the DBMS. Consequently, the overlap in the business logic identification is not redundant, but is an enforcement that strengthens our confidence in the business process and logic extraction.

4. Case Studies

To demonstrate the effectiveness of our proposed approach to identify the business logics, we performed case study on industrial e-commerce applications.

We analyze control conditions and entity behaviors to identify business data, policies and logics from the source code as discussed in Section 3. Furthermore, we generate complete trace records by applying static tracing technique that simulates the control flow of the source code [5] from the entry to the exit of the process (i.e. from *main* method of a program until this program terminates). The trace records contain the identified business policies and logics with the inputs and outputs as well as the execution sequence of the logics. One example of the extracted process, namely *Update TA Spending for Limit Check*, is shown in Figure 4 (simplified version of our original output).

```
1  <Decision expression="hasNext"/>
2  <Loop condition="yes" endline="236" startline="234">
3         <Task name="abRightToBuyTC.setInitKey_referenceNumber"/>
4         <Choice expression="strTCCurrency==null">
5         <Yes>
6                <Task name="abObligationToBuyTC.setInitKey_referenceNumber"/>
7         </Yes>
8         <Choice expression="bMultipleTradingIds">
9         <Yes>
10               <Decision expression="i<iabOrderItemArray.length"/>
11               <Loop condition="yes" endline="330" startline="320">
12               <Task name="dPurchaseAmount=dPurchaseAmount.add(getTaxAmountInEJBType())"/>
13               <Task name="dPurchaseAmount=dPurchaseAmount.add(getShippingChargeInEJBType())"/>
14               <Task name="dPurchaseAmount=dPurchaseAmount.add(getShippingTaxAmountInEJBType())"/>
15               <Task name="dPurchaseAmount=dPurchaseAmount.add(getTotalAdjustmentInEJBType())"/>
16               <Task name="convertMonetaryValue"/>
17               </Loop>
18        </Yes>
19        <No>
20               <Taskname="convertMonetaryValue"/>
21        </No>
22        <Choice expression="spendingLimit!=null">
23        <Yes>
24               <Task name="findTradingPurchaseTotal"/>
25               <Task name="findTradingRefundTotal"/>
26        </Yes>
27        </Choice>
28 </Loop>
```

Figure 4 – "Update TA Spending for Limit Check" Process Extracted from Implementation

Compared with our previous work in [4], the new approach is able to identify more business logics accurately. We were able to identify four business logics from *Update TA Spending for Limit Check* by the method discussed in [4]; the number of business logics is increased to ten. This improvement has shown the effectiveness of the new approach.

In Figure 4, the business logics located on Line 3, 6 and 12~15 are newly identified. Furthermore, the pre-derived condition of the CHOICE on Line 8 is a business policy for the reason that the same method *convertMonetaryValue* was invoked in both branches, as discussed in Section 3.2. However, before the aid of business data, this policy had little significance because both branches contained only an identical logic, namely *convertMonetaryValue* (Line 16 and 20), whereas the *Yes* branch actually performs a number of additional computations. By considering business data, we realize the significance of the computations and capture them as the business logics (Line 12~15).

As afore mentioned, we are able to prove that business data and policies play important roles in the business processes in three-tier architecture systems. By considering business data and policies, we can identify business logic and generate complete and precise business processes from the source code automatically.

5. Conclusion

The three-tier architecture defines explicit interfaces to the DBMS and indicates the input and output data for the business logics. It offers good starting points for the business process and logic extraction. We identified the business data from the database operations and the business policies from the behaviors of the objects and the calculation of the outputs. Combining the business data and policies together, we automatically located the business logics. By utilizing static tracing technique, we generated complete records that outline the interactions between business logics to form business processes. Finally, our case studies demonstrated the improved effectiveness of our new approach by doubling the number of the identified business logics and by extracting more precise computations performed from the source code.

References

[1] H. Sneed and K. Erdos, “Extracting Business logics from Source code”, in proceedings of 4th International Workshop on Program Comprehension, 1996

[2] D. C.C. Poo, “Explicit Representation of Business Policies”, in proceedings of Asia Pacific Software Engineering Conference, 1998

[3] M. Hung and Y. Zou, “A Framework for Exacting Workflows from E-Commerce Systems”, in proceedings of Software Technology and Engineering Practice 2005

[4] Y. Zou et al, “Model-Driven Business Process Recovery”, in proceedings of the 11th Working Conference on Reverse Engineering, 2004

[5] T. Eisenbarth et al, “Static Trace Extraction”, in proceedings of 9th Working Conference on Reverse Engineering, 2002.