A comparative evaluation of the three prominent approaches in adaptable software architecture

Roslinda Maznan and Wan Mohd Nasir Wan Kadir
Software Engineering Department
Faculty of Computer Science and Information Systems
Universiti Teknologi Malaysia
81310 UTM Skudai, Johor, Malaysia
e-mail: roslinda@auspac.com.my, wnasir@fsksm.utm.my

Abstract

Due to the inherent dynamic nature of the software environment, software evolution is inevitable. A large portion of total software lifecycle cost is devoted to introducing new requirements, and removing or changing the existing requirements. Many research projects attempt to find a more applicable way for building a software system that is flexible to changes. These efforts lead to the extensive study in software architecture that is adaptable to changes. In this paper, we compare three prominent approaches to adaptable software architectures namely Adaptive Object Model, Coordination Contract and Aspect Oriented Programming. It provides a brief description on the properties of each approach, and explains the comparative evaluation framework that is used in the evaluation process. Sejahtera System, which has a dynamically changing user requirement, is chosen as the case study to facilitate the consistent comparison of the selected approaches. We strongly believe that the results presented in this paper may provide a foundation in improving the state-of-the-art adaptable software architecture approaches.

KEYWORDS: Software architecture, adaptable software architecture, software adaptation, architecture-based evolution, software evolution.

1. Introduction

Nowadays, nearly all of commercial and government organizations are highly dependent on software systems. Due to the inherent dynamic nature of their business environment, software evolution is inevitable. The changes generated by business policies and operations are propagated onto software system. A large portion of total software lifecycle cost is devoted to introduce new requirements, and remove or change the existing requirements [1]. However, software evolution must be accomplished for the software to remain useful in its environment [2]. Due to this reason, software evolution is considered as a key research challenge in software engineering.

Many research projects attempt to find a more applicable way for building a software system that is flexible to changes as well as predicting the effect of requirements change [3]. Most of them adapt the existing well-proven widely accepted software technologies or design techniques. Developing solution on top of the existing technologies or proven techniques, such as object-oriented, expert system, distributed object, software architecture, design patterns, and metamodelling, may increase the chances for the approach to be accepted by software development community.

In this paper, we have selected three prominent approaches based on the exhaustive investigation on the state-of-the-art approaches in adaptable software architectures i.e. Adaptive Object Model, Coordination Contract, and Aspect-Oriented Programming. It starts with a brief introduction of each approach. Next, it explains the comparative evaluation framework that is used in the evaluation process. It is followed by the discussion of the evaluation results. Finally, it presents the conclusion and future direction of our research. For a more subjective and consistent evaluation, we have chosen the specification of the real system, namely a welfare management system, as the case study.

2.0 Architecture-based Software Evolution

Software evolution is the cumulative effect of the set of all changes made to a software system over its entire life-cycle [4, 5]. It concerns any change that is being made to the entire set of programs, procedures, and related documentation associated with a computer system that make up a software system [6]. The study of software evolution is important due to the change in customer requirement, need for new development of software, adding new software features, and fixing software defects during the maintenance phase of software life cycle [7]. Recently, there is interest in architecture-based software evolution [8-10].

In general, software architecture consists of components, connectors and organization of its components and connectors [5, 11, 12]. These architectural constituents can be manipulated and further defined to achieve an adaptable architectural design, which in turn improves evolvability of a software system. For examples, refining the role of connectors makes run-time evolution of software architectures feasible [13], and introducing good abstractions of the components for composition improves software evolvability [14]. In product-line architectures, i.e. a set of software systems that share core product architecture, the architectural constituents are carefully identified and defined for a future product member evolution [15].

This section presents three prominent approaches to adaptable software architecture i.e. adaptive object model, coordination contract and aspect-oriented programming. Among the leading approaches is *Adaptive Object Model* (AOM), which is defined as "a system that represents classes, attributes, and relationships as metadata" [16, 17]. It is a meta-architecture that allows users to manipulate the concrete architectural components of the model such as business objects and business rules. These components are stored in a database instead of code. Thus, a user only needs to change the metadata instead of changing the code to reflect domain changes. Simple rules such as defined types of entities, legal subtypes, relationships, and cardinality, are normally controlled by object-oriented modeling semantics. *Strategies* and *RuleObjects* are used to model complex rules. The example of AOM is show by the Figure 1 below.

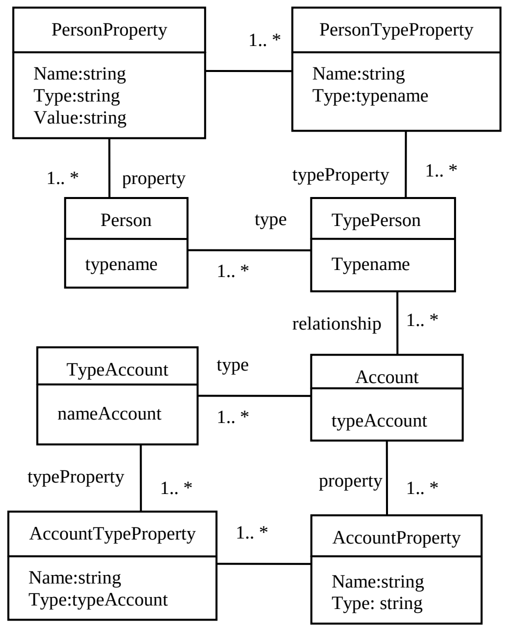

Figure 1: Example of Adaptive Object Model

Based on the example of *Adaptive Object Model*, there show combination two elements such as *TypeObject* and *ObjectProperty*, these combinations is also call as *TypeSquare*. There have showed how the property of the object is different between another object.

Coordination Contract aims to separate core business entities that are relatively stable and volatile business products that keep changing for the business to remain competitive [18]. Volatile business products are implemented as contracts. Contract aims to externalize the interactions between objects (core entities) by explicitly defining them in the conceptual model.

Here is given an example of Contract for openAccount:

```
Contract openAccount package
   Partners x:Customer; y:Account;
   Constants con_Balance:Integer
   Attributes  Balance:  Integer;  Name:String;
                  IC:String;
Invariants
   ?own(x,y)=TRUE
   y.AverageBalance() >= con_Balance;
Coordination
   openAccount: when x.calls (y.newReg(z))
   do y.newRegister(z)
   with y.balance()+Balance() > z;
end contract
```

Based on the example of *Coordination Contract* shown in Figure 2, there was exist three contract which is *OpenAccount*, *updateInformation* and *manageAccount*. There are mainly for establishing the interaction and coordination rule between the components to another component.

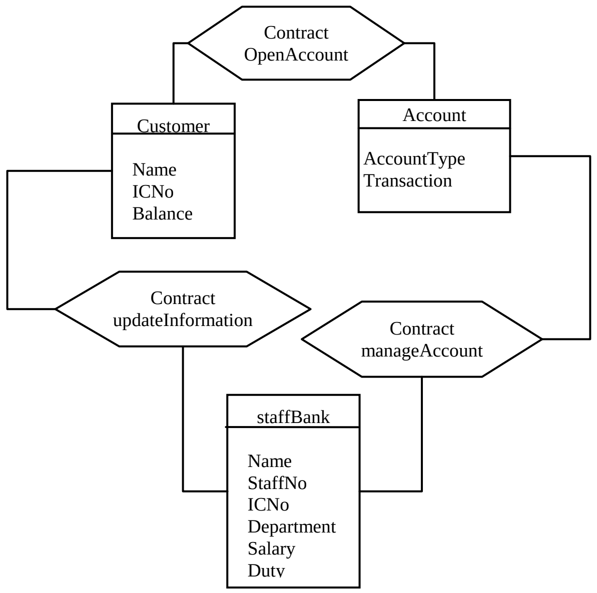

Figure 2: Example of Coordination Contract

Aspect-Oriented Programming (AOP) is a paradigm that enables separation of concern. It is a set of techniques that provides the means to add additional behaviour into existing classes and operations during compilation or execution [19] and provides a clean way of encapsulating crosscutting concerns [20]. Crosscutting concern is a concern that affect multiple classes. It is responsible for weaving or composing the different concerns into a coherent program. However, with aspect-oriented, the code implementing the security policy could be extracted out from all classes and consequently integrated into an aspect.

Aspect is a programming contruct in AOP that gives the ability to add class extensions into the existing classes. It includes pointcuts, advices, and intertype declarations. The example of the AOP design model is shown in Figure 3. The code example that uses the element of aspect is shown below:

```
public String DBTrans.countBalance(String);
public String DBTrans.openAccount( );

Aspect AccountBank {
 pointcut p( ): call (public String DBTrans.
    countBalance (String));
 pointcut q( ): call (public int
    DBTrans.openAccount( ));

before(String s ): p(s) {
 System.out.println(“Balance is - ” +s);
}

before(int i): q( ) {
 System.out.println(“New Account ” +i+ “is
    open”);
}
```

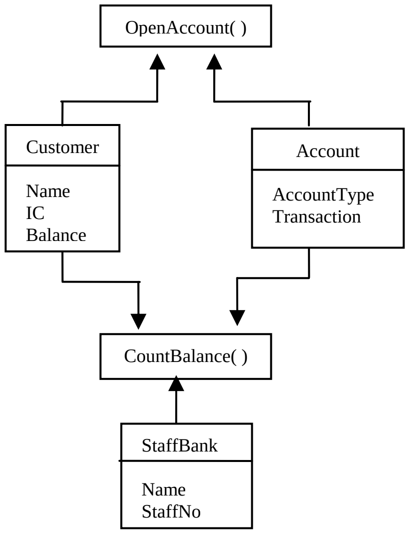

Figure 3: Example of Aspect-Oriented Programming

Based on the example of *Aspect-Oriented Programming*, there was exist two aspect which is *OpenAccount* and *CountBalance*. An aspect is extracting the object from different concern and integrates it.

3.0 The Comparative Evaluation Framework

In this section, we describe the evaluation framework that is used to compare the selected approaches. The evaluation framework consists of criteria that are classified into three components i.e. modelling language, process and pragmatic.

3.1 Modelling language

Modelling language is a set of symbols (either graphical or textual), syntax and semantics that is defined for supporting and representing the specified concepts of an approach. The criteria considered under the modelling language component include understandability, expressiveness, formality, and evolvability.

Understandability refers to the degree of adaptation of the modelling language representation. Natural language is highly understandable compared to formal language or programming language. However, structured natural language or pseudo-code is more understandable to system developers. *Expressiveness* is related to a capability of the expressions produced by the modelling language in completely and correctly presenting the adaptation concepts. *Formality* is the measure of rigour in the specification produced by a modelling language. It is important for the implementation, executability, testability, and preciseness of adaptive expressions. *Evolvability* refers to the flexibility of a software system in dealing with adaptation changes.

3.2 Process

Process is a series of well-defined steps or activities with corresponding input and output products which assist users (such as analysts, developers, and managers) to perform software development tasks. Lifecycle coverage, process description, coherence, and support for evolution are considered as the evaluation criteria in the process component.

Lifecycle coverage is a set of common development phases defined by the evaluated approach. These phases include analysis, design, implementation, and maintenance. *Process description* is how the availability of detailed descriptions about steps or activities within the scope of its lifecycle coverage. The description includes deliverables at each stage and guidelines for quality or project management. *Coherence* is the degree of logical connection from a flow of one-step to another step of the process. For the *support of evolution*, an availability of process description regarding the maintenance or evolution of software adaptation and the relevant software design components.

3.3 Pragmatic

Pragmatics is concerned with the practical aspects of deploying and using the approach. It includes both management and technical issues. Among the criteria associated to this component include usability, resource availability, and openness.

Usability is the easiness of applying the process and syntax. It is hard to use an approach if the modelling language syntax is too rigorous or too vague, or the process is too complex to be followed. The availability of resources such as texts book, user's group, and training are important for the users in facing their everyday problem in establishing their software development tasks. *Openness* is the degree of independence of the solution of the approach to certain implementation platforms such as architecture framework, paradigms, or programming languages.

4.0 The Comparative Evaluation Results

In our study, we compare the selected approaches to adaptable software architecture using the proposed evaluation framework. For a more systematic and consistent comparison, we use the specification from a real-world application namely Sejahtera System. Sejahtera System is a web-based software application that is used to manage the distribution of government aid to the needy families or individual. It functionalities include the registration, selection, and monitoring of the aid receivers.

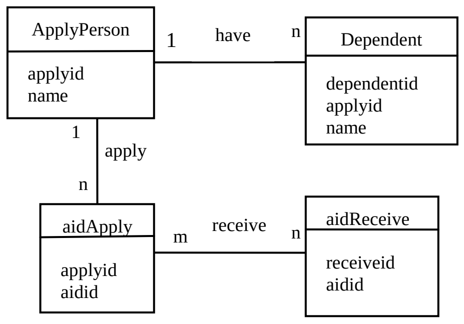

Figure 4: The example of Sejahtera System design using traditional OO approach

As a comparative evaluation baseline, we develop the design model for our case study using the traditional object-oriented approach. The design is partly shown in Figure 4. Based on this baseline, we develop the design models using the selected approaches and consequently compare each design model. Figure 5 shows the design model based on AOM approach.

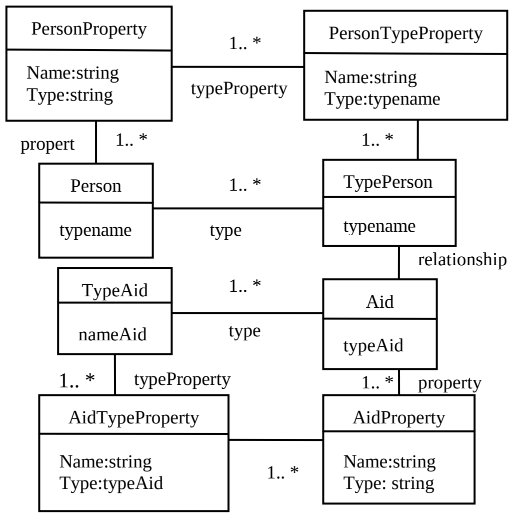

Figure 5: Sejahtera System design using Adaptive Object Model

As shown in Figure 5, there are eight classes in the AOM-based design model i.e. class Person, PersonProperty, PersonTypeProperty, TypePerson, Aid, TypeAid, AidProperty, and TypeAidProperty. It can be observed that the class Person in AOM model combines the classes ApplyPerson and Dependent in the traditional object-oriented model. Similarly, the class Aid represents the classes aidApply and aidReceive. The AOM model declares the classes, attributes, relationships and behaviours in terms of meta-data.

Regarding the CC model, we have identified three contracts for the same selected design part of Sejahtera System, which is shown in Figure 6.

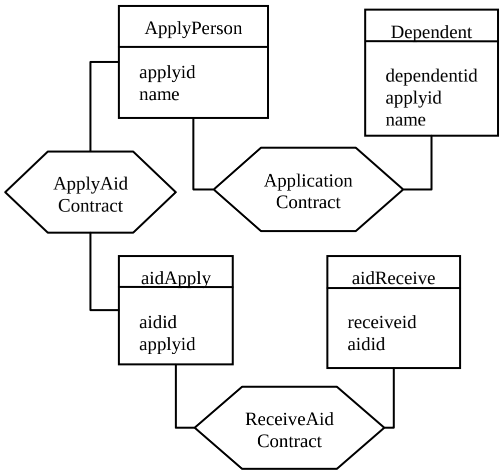

Figure 6: Sejahtera System design architecture using Coordination Contract

The specification of one of the contracts in the above CC model is given below:

```
Contract application package
  Partners x:ApplyPerson; y:Dependent;
  Constants AidValue:Integer;
     dependent:Integer, salary: money;
  Attributes dependent:Integer; Name:String;
     IC:String; salary: money;
  Coordination application :
     when x.calls(y.newReg(z))
     do y.newReg(z)
     with x. newReg( );
end contract
```

Relating to the AOP model, there are two classes of aspects created using AOP approach to provide the means to add additional behaviour into existing classes and operations during compilation or execution.

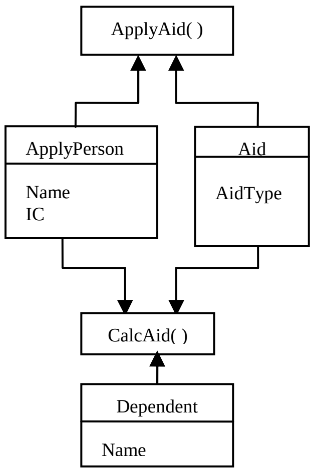

Figure 7: Sejahtera System design architecture using Aspect-Oriented Programming

A code snippet that shows the example of code used in aspect element is shown below:

```
public String DBTrans.CalAid(String);
public String DBTrans.ApplyAid( );
Aspect AidSejahtera {
   pointcut p( ): call (public String
      DBTrans.CalAid (String));
   pointcut q( ): call (public int
      DBTrans.ApplyAid ( ));
   before(String s ): p(s) {
      System.out.println(“Total of Aid are ”
            +s);
   }
   before(int i): q( ) {
      System.out.println(“New Aid ” +i+ “is
            apply”);
   }
```

From the comparison results between three adaptable software architecture approaches, it is found that AOM is more understandable than CC and AOP since it heavily uses the graphical representation and more simplified design model. However, AOM is less expresive due to its limited language expression. The CC model scores a high formality because of its rigorous contract specification.

In terms of the process criteria, it is observed that the AOP approach have the widest coverage in the software lifecycle whilst AOM provides less description on its software process. Regarding support for evolution, AOP gives the highest support by providing enough maintenance or evolution coverage in its software process.

Concerning pragmatic aspect, the AOM model is found to be less usable than others since it is hard to develop the meta-level of design model although the well-developed model is more understandable. In terms resource availability, there plenty of references and examples on applying the AOP approach due to its popularity.

Table 1: The comparative evaluation results of the three adaptable architecture approahes

Criteria \ Technique	AOM	CC	AOP
Modelling Language			
Understandability	High	Medium	Medium
Expressiveness	Low	Medium	Medium
Formality	Low	High	Medium
Evolvability	High	High	High
Process			
Lifecycle Coverage	D,I,M	D,I,T,M	A,D,I,T,M
Process Description	Low	Medium	Medium
Coherence	Medium	Medium	Medium
Support For Evolution	Low	Medium	High
Pragmatic			
Usability	Low	Medium	Medium
Resource Availability	Medium	Medium	High
Openness	Medium	Medium	Medium

Lifecycle coverage: A-Analysis, D-Design, I-Implementation, T-Testing, M-Maintenance/Evolution

6.0 Conclusion and Further Work

In this paper, we briefly describe the three prominent approaches to software evolution i.e. AOM, CC, and AOP. We also propose the comparative evaluation framework that consists of various criteria classified under three main components namely programming language, process, and pragmatic. Based on the evaluation results, we found that AOM is better than others in terms of modelling language. However, AOP is superior to others in process and pragmatic aspects. We hope that the results presented in this paper may provide a starting point for future research in improving the existing state-of-the-art adaptable software architecture approaches.

At the moment we are continuing our research in two directions. First, we aim to develop our own adaptable software architecture that extends and improves the existing approaches, and extensively apply it to the industrial-strength case study. Second, we attempt to develop the automated tools that support the proposed approach. The ultimate aim of our research is to produce an adaptable software architecture that supports software evolution and satisfies most of the criteria included in our comparative evaluation framework.

References

[1] P. Grubb and A. A. Takang, *Software Maintenance: Concepts and Practice*. Singapore: World Scientific Publishing, 2003.

[2] M. M. Lehman, "Laws of Software Evolution Revisited," in *European Workshop on Software Process Technology '96*, 1997.

[3] A. Finkelstein and J. Kramer, "Software Engineering : A Roadmap," in *Conference on the Future of Software Engineering*, Limerick, Ireland, 2000.

[4] E. Eijkelenboom, "Trends in Software Evolution," PhD Thesis, Utrecht University, 2005

[5] M. A. Babar, L. Zhu, and R. Jeffery, "A Framework for Classifying and Comparing Software Architecture Evaluation Methods," in *2004 Australian Software Engineering Conference (ASWEC'04)*, Melbourne, Australia, 2004.

[6] C. Varhoef, "Software Evolution: A Taxonomy."

[7] L. C. Nary Subramanian, "Software Architecture Adaptability: An NFR Approach."

[8] M. E. Fayad, H. S. Hamza, and H. A. Sanchez, "Towards scalable and adaptable software architectures," in *IEEE International Conference on Information Reuse and Integration*, 2005.

[9] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjorven, "Using architecture models for runtime adaptability," *IEEE Software*, vol. 23, no. 2, 2006, pp. 62-70.

[10] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng, "Composing adaptive software," *IEEE Computer*, vol. 37, no. 7, 2004, pp. 56-64.

[11] D. E. Perry and A. L. Wolf, "Foundations for the Study of Software Architecture," *Software Engineering Notes*, vol. 17, no. 4, October 1992, 1992, pp. 40-52.

[12] M. Shaw and D. Garlan, *Software Architecture : Perspective of an emergance discipline*. New Jersey: Prentice Hall Inc., 1996.

[13] P. Oreizy, N. Medvidovic, and R. N. Taylor, "Architecture-Based Runtime Software Evolution," in *International Conference on Software Engineering 1998 (ICSE'98)*, Kyoto, Japan, 1998.

[14] L. Andrade, J. Fiadeiro, J. Gouveia, and G. Koutsoukos, "Separating Computation, Coordination and Configuration," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 14, no. 5, 2002, pp. 353-59.

[15] M. Svahnberg and J. Bosch, "Issues Concerning Variability in Software Product Lines," in *Third International Workshop on Software Architectures for Product Families*, 2000.

[16] D. Riehle, M. Tilman, and R. Johnson, "Dynamic Object Model," Proceedings of the 2000 Conference on Pattern Languages of Programs (PLoP 2000), Technical Report WUCS-00-29, Dept. of Computer Science, Washington University, 2000.

[17] J. W. Yoder and R. Johnson, "The Adaptive Object Model Architectural Style," in *Proceeding of The Working IEEE/IFIP Conference on Software Architecture 2002 (WICSA3 '02)*, Montreal, Canada, 2002.

[18] L. Andrade and J. Fiadeiro, "Coordination Technologies for Managing Information System Evolution," in *13th Conference on Advanced Information Systems Engineering*, Interlaken, Switzerland, 2001.

[19] P.-W. N. Ivar Jacobson, *Aspect-oriented Software Development with Use Cases*: Addison-Wesley, 2004.

[20] L. Chen, "Aspect-Oriented Programming in Software Engineering, 2004."