INFORMATYKA EKONOMICZNA BUSINESS INFORMATICS 1(23) • 2012

ISSN 1507-3858

Cezary Hołub
Wrocław University of Economics
e-mail: cezary.holub@op.pl

THE IMPORTANCE OF AOP IN SOA ARCHITECTURE[1]

Abstract: Service oriented architecture (SOA) represents a model to build distributed applications by loosely integrating a number of web services. Aspect-oriented programming (AOP) is a programming paradigm that increases modularity by enabling the improved separation of concerns. Due to the large variety of potential users and the dynamic loosely coupling nature, web services are subject to frequent evolution, which is often required to be done rapidly, and multiple versions may co-exist. In reality it is often the case that existing web services do not perfectly match user requirements in target systems. To achieve the smooth integration and high reusability of web services, mechanisms to support automated evolution of web services in either functional or non- functional aspects are highly in demand. The aim of the paper is underlining the usability of AOP for SOA via theoretical divagation and practical examples in code. We will try to demonstrate the advantages when using the aspect approach in SOA. This article also briefly presented the same concepts of AOP and SOA.

Key words: AOP, SOA, Aspect-Oriented Programming, Service-Oriented Architecture, software architecture.

1. Introduction

All software systems, from standalone applications to complex distributed systems, evolve over time. Software engineering is facing a big challenge in maintaining reliable and efficient but constantly evolving systems. There is a paradigm shift from object oriented systems to service oriented systems. As more businesses are adopting cutting edge information technologies to provide cost effective dynamic services to their customers, the area of web services is becoming highly attractive and lots of research efforts are being applied to develop new software mechanisms. The underlying architecture of web services is based on the concept of Service Oriented Architecture (SOA). The main purpose of the development of SOA was to create systems which can provide inter-system communication and data exchange. This is achieved by integrating existing software to support processes that run on various software applications and to create a repository of reusable components that can be used in business processes in order to solve real business problems. The core idea

[1] This paper is an extended version of [Hołub 2012].

of SOA is to decouple functional units and expose them as independent services to other programs and thereby foster reuse. One of the biggest hurdles to achieving true service orientation is separating the needs of the service from the needs of the application. While services should only be concerned with the business functionality that they are exposing, it is not unusual for application specific features to "leak" into the service layer.

We can now overcome the difficulties that have prevented widespread adoption of aspects in practical implementations and enable architects to factor out cross-cutting concerns from the services they are designing. Cross-cutting concerns are requirements that are usually not of primary but of secondary interest. In other words, there is a second dimension in solving problems while building a program. But programming languages and the chosen design criteria are one dimensional. For example a program accesses a database. Let this be a primary requirement and, now, add logging as a secondary requirement. This means that every access to the database is supposed to be written to a file. The logging cross cuts whatever is done by this program because the database accesses are usually scattered all over the source code. As a consequence, implementing cross-cutting concerns with programming languages has been cumbersome.

One major tenet of software development practices has been the concept of *separation of concerns*. This tenet states that concerns or behaviors should be separated into well-defined code modules so that changes to these concerns are able to be made in one place. Therefore, AOP introduced a concept to collect these concerns, here called aspects, statements at a single place. Through that, a separation of concerns is achieved and changing an aspect only results in changes to a single file. But the statements need to be propagated back to the places they belong to. This distribution is done by a code weaver which inserts the aspect's source code at certain join points before the source code is compiled. For example, every access to a database is supposed to be logged. Therefore, calls of particular methods have to be monitored. Hence, all these methods are join points and the code weaver inserts a logging mechanism before these methods are called. In other words, the cross-cutting concern is realized as an aspect logging mechanism and is woven at compile time. The objective we are going to address is cross-cutting concerns in a SOA environment and we refer to these concerns as aspects as AOP does.

The current state of art we can find in [Sonchaiwanich et al. 2010; Krueger, Mathew, Meisinger 2006; Induruwana 2005].

2. Architectural and service aspects of SOA

Numerous vendors, application providers, system integrators, architects, authors, analysts firms, and standards bodies provide definitions of SOA. The definitions of SOA are diverse. Most are complementary and do not conflict with each other. SOA has a variety of definitions because the definition is often tuned to a specific audience,

as explaining SOA to a CEO is different from explaining SOA to a programmer. The term *service orientation* is often used synonymously with SOA, but just like SOA it has a wide range of interpretations. Service orientation is broader and represents a way of thinking about services in the context of business and IT [Holley, Arsanjani 2011].

We can say that service-oriented architecture (SOA) is an architectural style that modularizes information systems into services. You then orchestrate collections of these services to bring business processes to life. In a successful SOA, you can readily recombine these services in various ways to implement new or improved business processes. SOA is a logical evolutionary descendant of the software modularization techniques that began more than 50 years ago with the introduction of structured programming. SOA's novelty is that it gives you increased flexibility in the choice of implementation technologies and locations for the service providers and consumers. The abstracted service interfaces also enable providers and consumers to evolve independently as long as the interfaces remain stable. The benefits of a SOA derive primarily from a single characteristic: the stability of the service interface. This stability, relative to the overall rate of systems changes, isolates service consumers from changes in service implementations. This isolation limits the scope of changes and thus reduces the cost of subsequent changes. You derive a much larger benefit when you are able to reuse services exactly as they are. Reuse avoids the cost of re-implementing or modifying the functionality encapsulated in the service [Brown 2008].

Figure 1 shows the architectural style of SOA. In this scenario, a service consumer invokes or uses a service. The service consumer uses the service description to obtain necessary information about the provider service (e.g. account service) to be consumed. The service description provides the binding information so the consumer can connect to the service, and the description identifies the various operations (e.g. open or close an account) available from the provider service. A broker can be used to find the service using a registry that houses information about the service and its location [Holley, Arsanjani 2011].

In Figure 1, it is difficult to determine how the architecture style of SOA enables the strategic benefits of SOA, such as lowering the lifetime cost of an application or bringing faster time to market or making applications resilient to change. SOA as an architectural style often makes the SOA project solely an IT endeavor, where the strategic business benefits of SOA no longer become the focus or measured outcomes. The benefits of process flexibility, time-to-market savings, lower costs, and others can be achieved with SOA, but only if we holistically adopt all stakeholder views of SOA and its application and pursue SOA adoption accordingly. When pundits, architects, consultants, or executives define SOA as a pure technology play or as solely an architectural style, they relegate it to the realm of IT science projects, overhyped technologies, and a marketing strategy rather than a novel approach to building flexible business solutions [Holley, Arsanjani 2011].

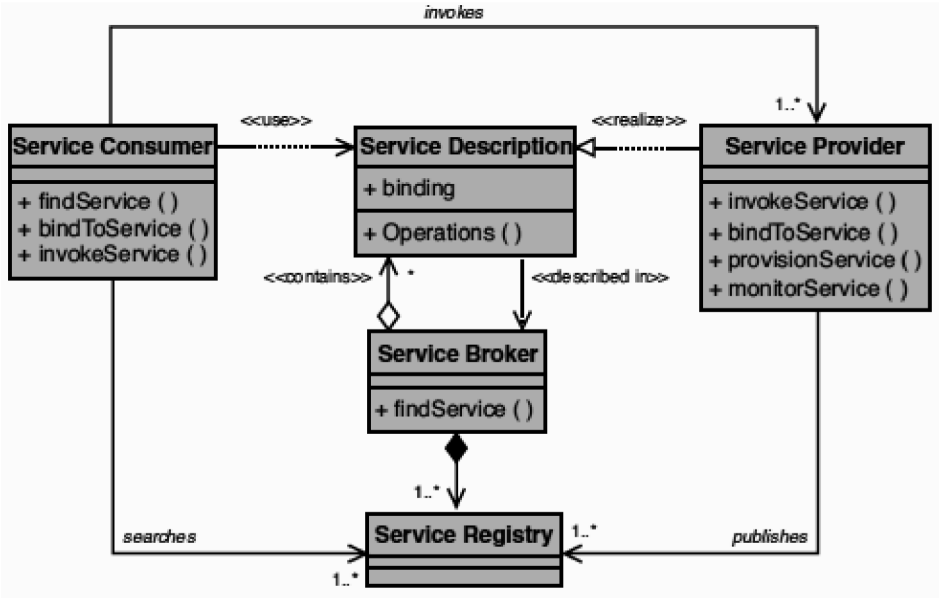

Figure 1. SOA as an architecture style

Source: [Holley, Arsanjani 2011].

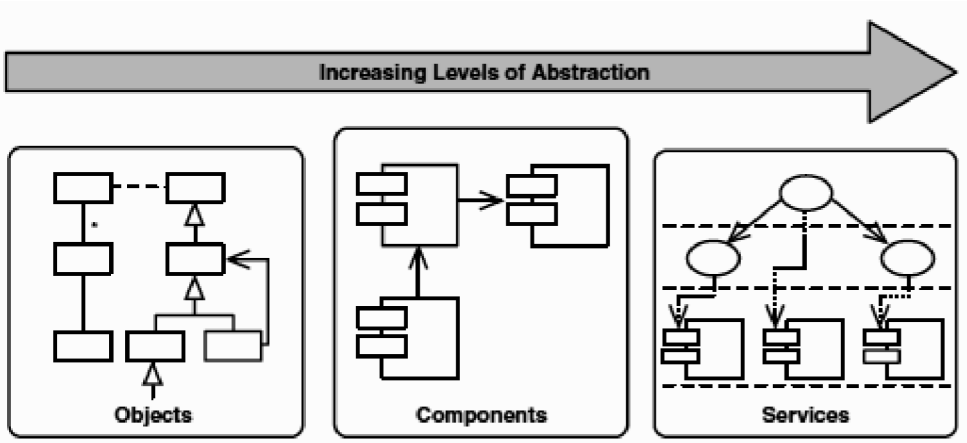

Figure 2. Levels of abstraction in SOA

Source: [Holley, Arsanjani 2011]

The most basic construct or building block of SOA is a service. Software engineering over the years has evolved from procedural to structured programming to object-oriented programming to component-based development and now to

service oriented. Figure 2 illustrates the different levels of abstraction from objects to services. Each evolution of abstraction builds on the previous, and SOA embraces the best practices of object and component development [Holley, Arsanjani 2011].

The architectural overview of SOA is on Figure 1, that illustration shows the fundamental constructs of SOA, such as the service consumer and the service provider and their relationship. The consumer invokes a service, the business functionality, by contract. The provider of the service defines the contract as a service description. An intermediary, such as a broker, uses a registry to find or search for published services. The service consumer, service provider, service description, service broker, and a registry are all part of the core of SOA. The following guiding principles define the ground rules for the development, maintenance, and usage of the SOA [http://wikipedia.org]:

- **Standardized Service Contract** – Services adhere to a communications agreement, as defined collectively by one or more service-description documents.
- **Service Loose Coupling** – Services maintain a relationship that minimizes dependencies and only requires that they maintain an awareness of each other.
- **Service Abstraction** – Beyond descriptions in the service contract, services hide logic from the outside world.
- **Service Reusability** – Logic is divided into services with the intention of promoting reuse.
- **Service Autonomy** – Services have control over the logic they encapsulate.
- **Service Granularity** – A design consideration to provide optimal scope and the right granular level of the business functionality in a service operation.
- **Service Statelessness** – Services minimize resource consumption by deferring the management of state information when necessary.
- **Service Discoverability** – Services are supplemented with communicative meta data by which they can be effectively discovered and interpreted.
- **Service Composability** – Services are effective composition participants, regardless of the size and complexity of the composition.

SOA realizes its business and IT benefits by utilizing an analysis and design methodology when creating services. This methodology ensures that services remain consistent with the architectural vision and roadmap, and that they adhere to the principles of service-orientation. Arguments supporting the business and management aspects from SOA are outlined in various publications.

3. Genesis and a general idea of AOP

In computing, aspect-oriented programming (AOP) is a programming paradigm which aims to increase modularity by allowing the separation of cross-cutting concerns. AOP forms a basis for aspect-oriented software development. AOP includes

programming methods and tools that support the modularization of concerns at the level of the source code, while "aspect-oriented software development" refers to a whole engineering discipline [http://wikipedia.org].

On the basis of procedural and object-oriented programming, already providing large modular projects, created and developed all the time aspect programming methodology. The goal is to improve the separation issues (separation of concerns), or separation of certain aspects of functionality (e.g. synchronization of access to resources, tracking progress of the program) into separate, independent modules. It is desirable at the same time to reflect a more modular system, the way we think about the problem rather than the method imposed by the tools. In complex systems, object-oriented programming does not allow for the modularization of all aspects of the system. Some problems will always cross the borders (called crosscutting) of the other modules. The idea of aspect programming is to provide mechanisms to a more complete modularization of the system – it is to simplify code, increase the speed of its creation, make it easier to understand, develop, maintain and reuse. Modularized intersecting issues (called crosscutting concerns) are called aspects [Colyer et al. 2009].

In general opinion, the main creator of the AOP shall be deemed to Professor Gregor Kiczales from the University of Vancouver. He finished his work in the nineties and it has become the basis for the emergence of this method and the formulation of its principles. For the first time he used the term AOP in 1996 during his tenure at Xerox Corporation. This method met with great interest and initiated several studies on its use in various fields, as well as some conference appearances. The article "Aspect-Oriented Programming" written by Gregor Kiczales is considered to be the first publication about aspect programming [http://www.parc.com/research/projects/aspectj/downloads/ECOOP1997-AOP.pdf].

The key concepts associated with aspect programming are [Jacobson 2004]:

– **Aspect** – a modularization of a concern that cuts across multiple classes/units/objects/components/services (Figure 3).
– **Concern** – this is a question or problem that must be resolved to achieve the objective
  • functional,
  • non-functional – concerns that crosscut are referred to as crosscutting concerns in AOP terminology and aspect can be used to encapsulate them.
– **Cross-cutting concern** – this is a problem whose representation is dispersed along the representation of other issues in decomposition of hierarchical system. The existence of the relationship between cutting issues may be dependent on the choice of the dominant criterion for decomposition. On this ground have grown both modularization and object-oriented methods (Figure 3.)
– **Join point** – a point during the execution of a program, such as the execution of a method or the handling of an exception.

- **Advice** – action taken by an aspect at a particular join point.
- **Pointcut** – a predicate that matches join points. Advice is associated with a pointcut expression and runs at any join point matched by the pointcut (for example, the execution of a method with a certain name).
- **Weaving** – linking aspects with other application types or objects to create an advised object. This can be done at compile time (using the AspectJ compiler, for example), load time, or at runtime. Spring AOP, like other pure Java AOP frameworks, performs weaving at runtime. In another words weaving is the process of composing a core functionality model with aspects and creating the final working system.

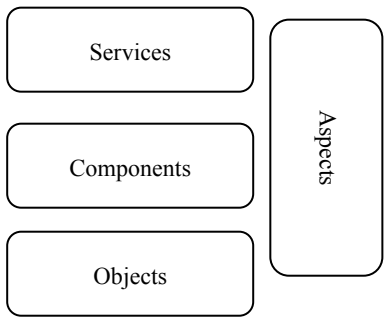

Figure 3. A layered organization of Objects, Components, Services and Aspects

Source: own preparation.

The main difference between object oriented programming (OOP) and aspect programming is not a different purpose (in both cases the grouping of similar concepts and different separation), but the other selection tools. In the case of object-oriented programming fundamentals these are: the concept of class, the encapsulation and inheritance. Typically, they allow the grouping of concepts used by one criterion, which is sufficient in most applications. The advantage of object-oriented programming is a strong and stable position on the market, and support in a wide range of popular programming languages. AOP therefore is not in conflict with the objectives or the object-oriented programming tools. We can say that the AOP approach is a superset of object-oriented. An example of such a situation is also part of the bank system (shown in Figure 4). The object/class Bank holds a collection of bank products (classes): Accounts, Credits, Deposits, Report. It is easy to see that each method of these classes needs common properties which are not connected with functional object division e.g. transact methods invocation. A trial of implementing such a feature by invoking in all classes source code which realized it, might result in disorder and reduce code maintenance.

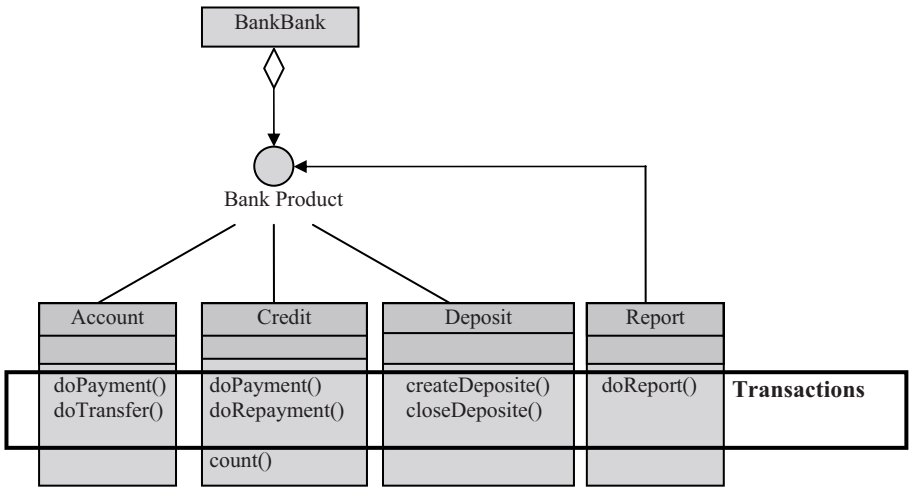

Figure 4. Part of bank system with using AOP approach, Transaction is an aspect

Source: own preparation.

One of the solutions for the practical use of the idea of aspect programming is AspectJ programming language. AspectJ is a language description of the aspects in the form of a superstructure built on Java. The first public version of the language was published in 2001, and that date is considered as the beginning of its existence. It is a versatile AOP extension of Java, which means that each Java program is also valid AspectJ language program. It does not modify any of the construction of the language, and adds new – especially the notion of aspect. Aspect is a special kind of class, which along with this class allows modularizing program.

4. Usability of AOP in SOA

As web services are geographically distributed and heterogeneous by nature, they are very volatile systems and are exposed to exceptions such as software or machine failure. This may lead to the unavailability of services. In such cases a replacement is needed to keep the high level of service intact at all times. Keeping replacements ready at all times is an infeasible scenario. One alternative was to make these services dynamic so that services are selected by a distinct module on the basis of their characteristics and availability. Further research has been conducted on the dynamic selection and integration of web services. This raised the issue of nonfunctional concerns. Nonfunctional concerns are constraints that cut across various sections of a business process. These research efforts led to the development of aspect oriented programming (or development). It has the benefits of modeling, encapsulating, and extraction of concerns. With the use of Aspect Oriented Programming techniques,

aspects could be applied to SOA to handle crosscutting concerns. The application of AOP with SOA in the development of web services is a current active area of research and has resulted in various middleware systems being developed to handle aspects with SOA [Wang, Bandara, Pahl 2010; Krueger, Mathew, Meisinger 2006].

There are some concerns, however, that have traditionally been difficult to isolate to classes. These concerns are referred to as *cross-cutting*, in that they are spread across several code modules. Some classic examples of these concerns are logging and security, which are traditionally spread throughout the application code. Figure 5 illustrates this concept. The figure shows logging and security as packages that cross-cut all layers in an N-tier system.

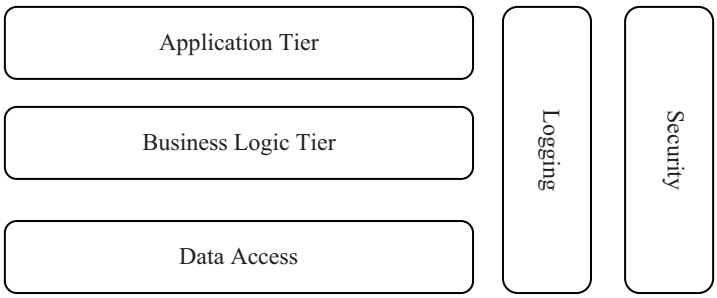

Figure 5. Cross-cutting concerns in a layered Architecture in N(3)-Tier system

Source: own preparation.

The ideal aspect is "orthogonal" to the target functionality, meaning that nothing the aspect does will affect the rest of the system. For example, caching and authorizing are powerful aspects, but you would not want your caching aspect to operate before your authorization aspect. Aspects in typical implementations have visibility problems. Aspects can be deployed throughout an application. In many cases when looking at a code one cannot infer whether aspects will be involved in its execution. Since aspects are applied orthogonally, they cannot be aware of their execution context. Different contexts would have widely different needs, but the aspect cannot know that. To solve the orthogonality problem, we deploy aspects in ordered sets. This allows for the scope and interaction of all participating aspects to be understood and validated ahead of time (Figure 6).

By relaxing the orthogonality constraint, we can use a wide variety of aspects including: instrumentation, authorization, caching, validation, exception handling, impersonation, transaction management, etc. This suite of behaviors represents a great deal of code that simply does not need to be written.

Web services is the most important technology available nowadays to implement Service oriented architectures (sometimes web services and SOA are equal terms). Web services architecture is almost "aspects-ready":

– Automated generation of stubs and skeletons (excellent points for advice),
– Separate standards to define non-functional properties (typical crosscutting concerns),
– Intermediate nodes that can operate on SOAP headers (also potential points for advice).

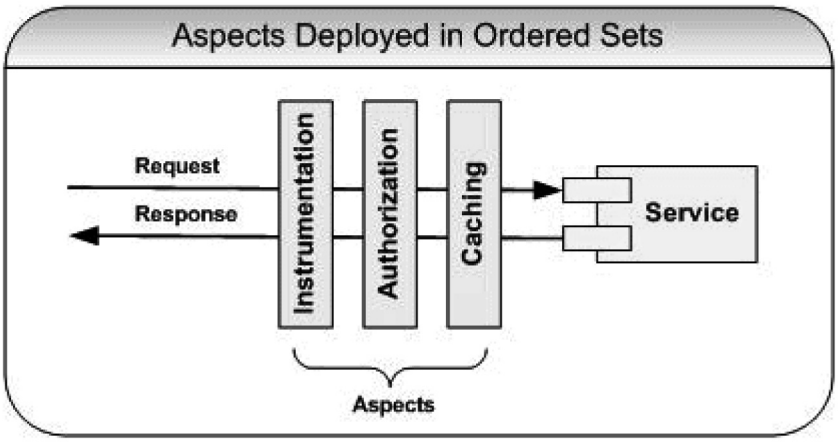

Figure 6. Aspects deployed in ordered sets

Source: [*Enabling Aspects*…].

We can use AOP to deal with crosscutting concerns existing in SOA like: validation, exception management, caching, logging, instrumentation, authentication and authorization. In the listing below we can see entangled[2] code to invoke a web service. This code can be divided into aspects (concerns):

In this example we will describe the realization of ideas and benefits of using aspect oriented approach combined with SOA security. When mission-critical logic functionalities are exposed as Web services in the SOA enabled system, greater security risks are also introduced to the enterprise entities. The system and resources such as services and data need to be protected from threats such as unauthorized access that may be imposed on the system. Using Aspect-oriented programming (AOP) as the software architecture pattern for SOA security implementation permits a practical means to "remove security logic and policy from application code completely" [Patel, McRoberts, Crenshaw 2009].

In order to pass user's credentials through a SOA composite web service, a security code is required to be included in the web service implementation. We developed an identity propagation framework to encapsulate the process of

[2] The implementation of a crosscutting concerns with non-AOP results in code tangling. Such that the code for a particular concern becomes intermixed with the code for another concern.

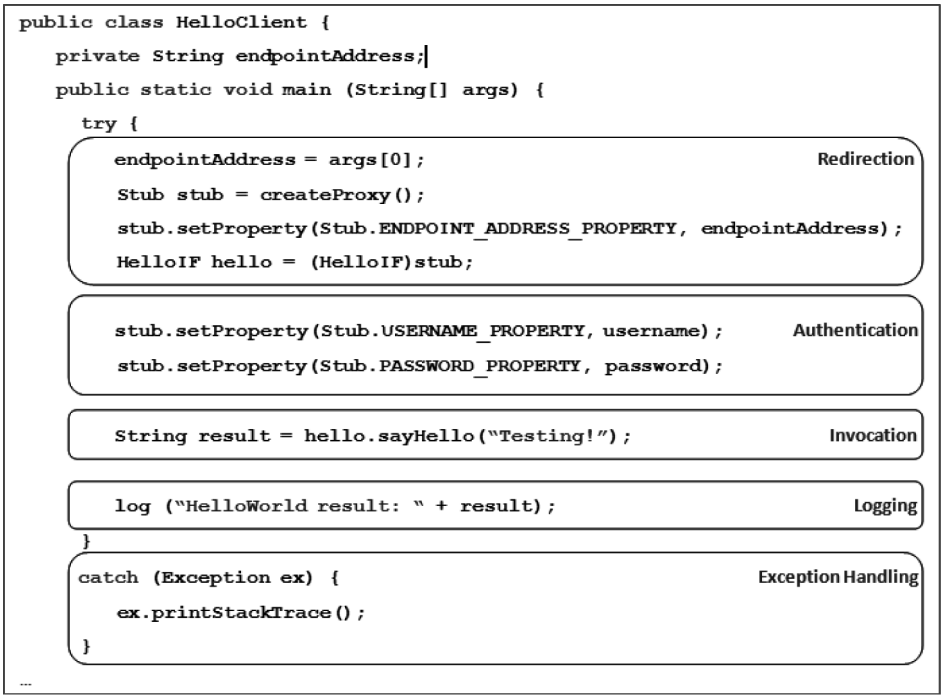

Figure 7. Example code with aspects (concerns)

Source: own preparation.

passing user credentials between services. The framework configures inbound and outbound SOAP handlers around a web service to transparently manage a SAML (Security Assertion Markup Language) security token, which is transmitted to subsequently invoked services. The rationale for such identity credential propagation framework has been well-articulated in a previous publication [Patel, McRoberts, Crenshaw 2009]. Figure 8 demonstrates this framework in detail:

In our case study, the application of this framework established a division in roles between application developers and security subject matter experts. Software developers were no longer tasked with implementing entire security policies in application code. Instead, their responsibility was reduced to implementing a smaller portion of security code to invoke the identity-propagation framework. The responsibility of managing the security policies was shifted to a specialized development team focused on managing the framework. This shift in responsibilities is important because application developers typically lack understanding of concerns outside of core business logic. While the division of roles was present, application developers still required some knowledge of the security framework, and could potentially affect its operation. This principle is illustrated in Fig. 9. Ensuring that the

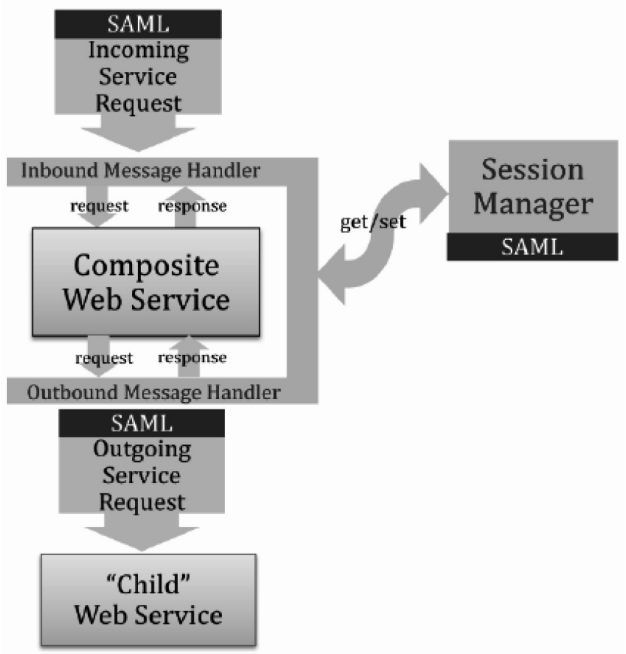

Figure 8. An identity credential propagation framework

Source: [Sonchaiwanich et al. 2010].

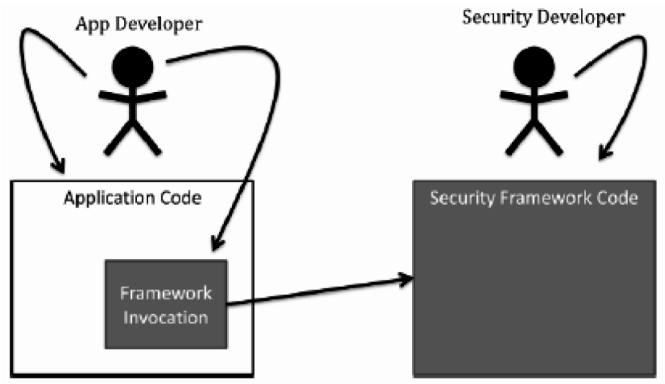

Figure 9. Invoking a security framework from the application code

Source: [Sonchaiwanich et al. 2010].

framework invocation code is implemented correctly requires involvement or review by the security framework developers. This does not fulfill the goal of independent verification for the application code and security code [Sonchaiwanich et al. 2010].

Figure 10 shows the code required to invoke the security framework. The *@HandlerChain* annotation configures an inbound SOAP handler to pre-process incoming requests. An outbound SOAP handler is configured by calling *setHandlerResolver*, which is required to be called on any subsequently invoked (outbound) services. Though the statements are seemingly simple, they are entry points into the framework and credentials would not be passed without them. Correct implementation of this code was the responsibility of application developers, as it was in-line with application code.

```
// Configure security framework:
// Sets up inbound SOAP handler for obtaining SAML
@HandlerChain(file="handler-chain.xml", name = "JAXWSIPService")
public class JAXWSIPServiceImpl implements JAXWSIPPortType {
    ...

    // Setup web service
    DiscoveryService serv = new DiscoveryService(wsdlLocation, serviceQname);

    // Configure security framework:
    // Sets up outbound SOAP handler for passing SAML
    serv.setHandlerResolver(new HeaderHandlerResolver());

    ...
}
```

Figure 10. Security handler code implemented within application business logic

Source: [Sonchaiwanich et al. 2010].

```
public aspect IdentityPropFramework {
    ...

    // Add the @HandlerChain annotation
    declare @type: JAXWSIPServiceImpl: @HandlerChain(file="handler-chain.xml");

    // Code insertion point
    public pointcut ServiceCreated(): ... ;

    // After the service is created configure the outbound SOAP handler
    after() returning (DiscoveryService service): ServiceCreated() {
          // inserts this code after pointcut
          service.setHandlerResolver(new HeaderHandlerResover());
    }
}
```

Figure 11. Security handler code encapsulated in an aspect

Source: [Sonchaiwanich et al. 2010].

While this approach helped in separating the application code from security code, we recognized this approach did not achieve the full separation of developer roles. We addressed this issue by applying AOP to complete the separation of application and security code. The code written to invoke the security framework was refactored into aspects as shown in Figure 11.

These aspects were then re-introduced into the core application through a process called weaving. The isolation of this code to aspects allowed application development to occur independently of security development. In addition, the application developers are unaware of the use of the framework, and are unable to affect the process of identity propagation. As shown in Figure 12, under this scheme responsibilities are truly divided and totally independent of one another.

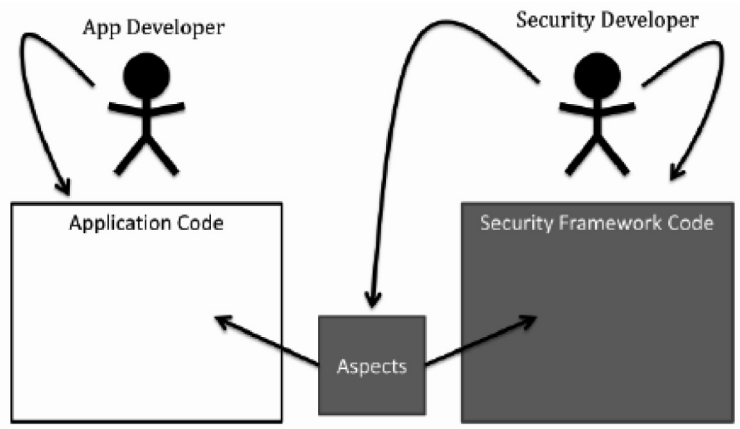

Figure 12. Invoking the security handler code from an aspect

Source: [Sonchaiwanich et al. 2010].

5. Conclusions

Despite the increasing popularity of SOA, the current state of art is that the service consumer has to use the web services as they are because of the lack of adequate web service evolution mechanisms. Aspect-oriented programming (AOP) provides an effective approach to manage the separation of business (mission) logic from the security concerns in software design and development to achieve business agility and software modularity in a SOA. In most cases, web services (SOA) are subject to frequent evolution requirements and multiple versions need to co-exist due to the large diversity of potential users. The evolution may occur to both functional and non functional aspects of a web service and requires to be done rapidly at low cost. The proposed approach applies aspect-oriented adaptation to the underlying components of a web service to meet the evolution requirements of the web service so that the

web service can be smoothly integrated into the target application. Automation and aspect-oriented deep level adaptation are the benefits of this approach. Such an approach enables web service developers to adapt their published web services to meet the integration requirement of specific web service applications. Our case studies have shown that the approach and tool are promising in their ability and capability to meet the evolution requirements of web services.

We illustrated how an IT system using AOP can successfully separate the none functional concerns, (e.g. security logic) from the business implementation in composite SOA services and applications. Ultimately the approach of combining AOP and SOA can reduces the costs on SOA application development, integration, certification, accreditation, and deployment, as well as helping to achieve business agility. A good starting point for future divagation in this area is to analyze "costs" in real existing IT systems made with AOP and SOA. There is an increasing role of AOP in SOA via deeper adaptability, higher automation and therefore smooth web service composition and wider reusability. In consequence, the target web service oriented systems will have better quality and more suitable functionality.

References

Brown P.C., *Implementing SOA: Total Architecture in Practice*, Addison-Wesley Professional, Boston 2008.

Charfi A., Mezini M., Aspect-oriented Web service composition with AO4BPEL, [in:] *The European Conference on Web Services*, Erfurt 2004.

Colyer A. et al., *Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools*, Addison-Wesley Professional, Boston 2009.

Courbis C., Finkelstein A., Weaving aspects into Web service orchestrations, [in:] *3rd International Conference on Web Services*, Orlando 2005.

Enabling Aspects to Enhance Service-Oriented Architecture, http://msdn.microsoft.com/en-us/library/bb245663.aspx.

Ermagan V., Krueger I., Menarini M., Aspect oriented modeling approach to define routing in enterprise service bus architectures, [in:] *30th International Conference on Software Engineering*, Leipzig 2008.

Ganser A., Hurtz S., Lichter H., A server side SOA meta model for assigning aspect services, [in:] *11th International Conference on Model Driven Engineering Languages and Systems*, Toulouse 2008.

Hewitt E., *Java SOA Cookbook*, O'Reilly, Sebastopol, USA 2009.

Holley K., Arsanjani A., *100 SOA Questions Asked and Answered*, Prentice Hall, Boston 2011.

Hołub C., AOP and SOA, are they fit?, *IADIS International Conference e-Society*, Berlin 2012.

Induruwana C., Using an Aspect Oriented Layer in SOA for Enterprise Application Integration, [in:] *3rd International Conference on Service-Oriented Computing*, Amsterdam 2005.

Irmert F., Meyerhoefer M., Weiten M., Towards runtime adaptation in a SOA environment, [in:] *21st European Conference on Object-Oriented Programming*, Berlin 2007.

Jacobson I., *Aspect-Oriented Software Development with Use Cases*, Addison-Wesley Professional, Boston 2004.

Josuttis N.M., *SOA in Practice*, O'Reilly, Sebastopol, USA, 2009.

Kongdenfha W., Motahari-Nezhad H., Benatallah B., An aspect-oriented approach for service adaptation, [in:] IEEE Transactions on Services Computing, Sydney 2009).
Krueger I., Mathew R., Meisinger M., Efficient exploration of Service-Oriented Architectures using Aspects, [in:] *28th international Conference on Software Engineering*, Shanghai 2006.
Laddad R., *AspectJ in Action Practical Aspect-Oriented Programming*, Manning, Greenwich, USA, 2003.
Lawler J. P., Howell-Barber H., Service-Oriented Architecture – SOA Strategy, Methodology, and Technology, Auerbach Publications, Boca Raton, USA, 2008.
Maciel R., David J., Oei M. Bastos A., Menezes L., Supporting awareness in groupware through an aspect-oriented middleware service, *Journal of Universal Computer Science* 2009, Vol. 15, No. 9.
Mcheick H., Mili H., Sadou S., El-Kharraz M., A comparison of aspect oriented software development techniques for distributed applications, [in:] *13th International Conference on Software Eng. and its Applications*, Paris 2000.
Miles R., *AspectJ Cookbook*, O'Reilly, USA, 2005.
Patel A., McRoberts M., Crenshaw M., Identity propagation in N-tier systems, [in:] *28th International Conference MILCOM*, Boston 2009.
Pulvermueller E., Klaeren H., Speck A., Aspects in distributed environments, [in:] *Proceedings of GCSE*, Erfurt 1999.
Rosen M., Lublinsky B., Smith K.T., Balcer M.J., *Applied SOA Service-Oriented Architecture and Design Strategies*, Wiley Publishing, Indianapolis 2008.
Sonchaiwanich E., Zhao J., Dowin C., McRoberts M., Using AOP to separate SOA security concerns from application implementation, [in:] *Military Communication Conference – MILCOM 2010*, San Jose 2010.
Svirskas A., Courbis C., Molva R., Bedzinskas J., Compliance proofs for collaborative interactions using Aspect-Oriented Approach, [in:] *4th IEEE International Conference on Services Computing*, Salt Lake City 2007.
Wang M., Bandara K., Pahl C., Distributed aspect-oriented service composition for business compliance governance with public service processes, [in:] *5th International Conference on Internet and Web Applications and Services*, Barcelona 2010.

Websites

http://wikipedia.org.
http://www.parc.com/research/projects/aspectj/downloads/ECOOP1997-AOP.pdf.

ZNACZENIE AOP W ARCHITEKTURZE SOA

Streszczenie: SOA to koncepcja tworzenia systemów informatycznych, w której główny nacisk kładzie się na definiowanie usług, które spełnią wymagania użytkownika. Pojęcie SOA obejmuje zestaw metod organizacyjnych i technicznych mający na celu lepsze powiązanie biznesowej strony organizacji z jej zasobami informatycznymi. Programowanie zorientowane aspektowo (AOP) to paradygmat programowania, który zwiększa modularność oprogramowania umożliwiając lepszą separację zagadnień. Ze względu na dużą różnorodność potencjalnych użytkowników i dynamiczne, luźno powiązane systemy, usługi sieciowe (ang. *web services*) podlegają ewolucji. Zmiany trzeba robić bardzo szybko, a różne wersje muszą współistnieć obok siebie. W rzeczywistości często jest tak, że istniejące usługi sieciowe nie odpowiadają w pełni wymaganiom użytkowników w systemach docelowych. Aby osiągnąć

płynną integrację, wysoką spójność i używalność usług sieciowych, bardzo pożądane jest posiadanie w miarę automatycznego mechanizmu rozwoju usług sieciowych. Mechanizm ten powinien wspierać wymagania funkcjonalne niefunkcjonalne systemu. Celem artykułu jest podkreślenie użyteczności AOP dla SOA, poprzez teoretyczne rozważania i przykłady praktyczne, z użyciem fragmentów kodów źródłowych aplikacji. Postaramy się wykazać zalety zastosowania aspektowości w architekturze SOA. Ten artykuł również pokrótce przedstawi same pojęcia AOP i SOA.

Słowa kluczowe: AOP, SOA, programowanie zorientowane aspektowo, architektura zorientowana na usługi, architektura oprogramowania.