Software development methods: SOA vs. CBD, OO and AOP

Mika Koskela, Mikko Rahikainen, and Tao Wan

Helsinki University of Technology
mkoskel@cc.hut.fi,mikko.rahikainen@iki.fi,simonwantao@yahoo.com

Abstract. Before the emergence of Service-Oriented Architecture (SOA), similar issues have been addressed by other software development paradigms including e.g. Object-Oriented Programming (OOP), Aspect-Oriented Programming (AOP) and Component-Based Development (CBD). In this study, these approaches are compared to SOA and their relationship is discussed. Different kind of criteria is used: methods of enabling software reuse, impacts on enterprise applications, architectural patterns and support for application extensions and versioning. It is concluded that many of the mechanisms provided by SOA can be enabled by software elements at the lower level. The main contribution of SOA is the focus on high-level business logic. SOA, CBD and OOP can be built on each other in layered manner with possible adoption of AOP having its effects on each level. Still, some things such as version control pose problems for each of the approaches.

Key words: Service-Oriented Architecture, Object-Oriented Programming, Component-Based Development, Aspect-Oriented Programming, SOA, OOP, CBD, AOP

1 Introduction

Service Orientation is currently one of the most appraised paradigms in software engineering and enterprise IT. However, many of its claimed benefits have been addressed by previous approaches in software development. Additionally, software systems are not built using services as the only element of construction but lower level artifacts are also required. These can be represented by components or objects, for example. In this report, the Service-Oriented Architecture and Computing (SOA / SOC) approach is compared to other software development paradigms. The domain of investigation is software development and this is the perspective that is taken on SOA here. In the comparison, the main emphasis is on Component-Based Development (CBD) and Object-Oriented Programming (OOP) approaches but also Aspect-Oriented Programming (AOP) is discussed. The objective is to address the main aims and benefits of these approaches. These are then compared to the characteristics of the Service-Oriented approach and differences and similarities in both means and ends are pointed out. Additionally, the relationship between these paradigms is described and their fitting together briefly explained.

This report is solely based on a literature study. In the study, the emphasis was on rudimentary literature discussing the approaches. This means that reviews on products and specific technologies as well as detailed implementation case studies were left out of the scope of the research. However, the aim was also to assess how well the paradigms have been currently realized. In the comparison of the approaches, specific frameworks of criteria are used.

The report is structured as follows: first section 2, each focal paradigm is briefly introduced and the approach of software reuse discussed. Next, in section 3, the approaches are compared with one another. There are three angles to this: level of abstraction, reuse, implementation, logical layering and support for application extension. Finally, in section 4, the findings of these comparisons are discussed and the conclusions are made.

2 Background

2.1 Service-Oriented Computing and Architecture

Papazoglou and Georgakopoulos [14] define Service-Oriented Computing (SOC) as a paradigm that utilizes services as fundamental elements of distributed application development. Accordingly, services are self-describing, open software components that communicate with each other over the Internet via public interfaces. Service-Oriented Architecture can be seen as an architectural concept based on SOC. Erl [3] recognizes the following characteristics for SOAs:

– *Loose Coupling*: minimum amount of interdependencies between services
– *Service Contract*: the interaction between services is based on a communication and document agreement
– *Autonomy*: services control the logic they encapsulate.
– *Abstraction*: services hide their internal logic from the external environment except those parts that are described in the service contract.
– *Reusability*: dividing logic to services promotes reuse.
– *Composability*: services can be assembled from other services
– *Statelessness*: minimum amount of activity specific information is stored.
– *Discoverability*: the existence of discovery mechanisms so that services can be discovered by their users

2.2 Object Oriented Programming and Development

Object oriented programming traces back to Simula language of 60s which was developed for describing and programming simulations.[17, Part III] Late 80s and 90s can be seen as a time frame when OOP moved to mainstream application development. Since then most influential OO-languages in industry have been C++ and Java. As a result of research and development in OO many tools and technologies have been introduced to support OO-based development as modeling languages, application servers, OO-databased or relational mapping tools and OO-based development processes.

In object oriented languages the execution flow of programs is passing messages between objects which represents concepts of the problem domain. Each object encapsulates related data or state of the object and operations in one unit, compared to functional programming where data is usually passed as a parameter to a function. This encapsulation of data allows code reuse and transparent change of implementation. Grouping objects to classes and allowing subclasses to inherit both interfaces and implementation from their parent classes allows on the other hand building on existing code.

Mapping persistent state of objects to relational database has been one major technical problem in OOP. Handling persistent objects and transactions is not a simple task and development wise it is usually better to use existing solutions - applications servers from both OSS and software vendors have been made to allow developers of business applications to concentrate more on business logic. Also there is OO-middleware for distributed development like CORBA ORB-implementations and, JRMI and other messaging solutions that allow remote methods calls between objects on different nodes.

Object oriented modeling is usually the first step in object oriented development model. From high level conceptual model of problem domain and use cases, the design phase continues eventually to lower level implementation view of classes and objects. Modeling of these aspects is usually carried out in standard modeling language such as UML. Direction have been towards model driven architecture, MDA, where actual program code can be compiled from modeling effort[12].

Key promises of OO-based development can be listed as increased code reuse, easier building on existing code, better change tolerance and decrease in errors by data hiding and encapsulation and finally OO can be seen as natural way to model the problem domain.

2.3 Component Based Development

A software component has been described as "a nontrivial, nearly independent and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture" by Brown and Wallnau. It is able to inter-communicate with other existing components with a predefined service. Clemens Szyperski and David Messerschmitt present the following five principles that a software component should have:

– Multiple-use
– Non-context-specific
– Composable with other components
– Encapsulated
– A unit of independent deployment and versioning [23]

Unlike objects in the Object Oriented Programming (OOP), a component is an "OOP objects composed", an object made to a specification. It does not concern what the specification is, such as, COM (Component Object Model), DCOM (Distributed COM, JavaBeans and etc., as long as the object clings to the specification. This enables a component to obtain features like reusability, reuse quality and so on. In practice, components are built on top of many software "objects" (although they are not confined to OOP) and provide a coherent unit of functionality. "All "objects" work together to performance a specific task at a particular level of service. " [18]

The idea of software componentized, built from prefabricated components, was first introduced in Dougalas McIlroy's address at the NATO conference on software engineering in Garmisch, Germany 1968 titled Mass Produced Software Components. The modern concept of a software component was wildly defined by Brad Cox of Stepstone, who called then Software ICs and created an infrastructure in order to market for these components by inventing the Objective-C programming language. [23]

After mentioning what the software component is, we are going to investigate CBDE(Component Based Software Engineering) and CBD (Component Based Development). As well know, when you purchase some PC components from any market vendors to set up a computer, it is very easy to assembly. You do not need to build PC system from hundreds of discrete parts, because each component has been designed to compatible each other and standardized. CBSE also does the same thing in order to design and construct systems by using reusable software components.

"Making applications from software components had been a dream in software engineering community since its very early time. Programmers have relied on the reuse of code and data structures from as far back as 1968, when M.D.McILROY published a paper titled "Mass-Produced Software Components"." [18]

The purpose of CBSE is to improve QoS, productivity and time-to-market in software development. It encourages the compositions of software system and focus on reusing and adapting existing components.[18]

There are three stages should be done in Components Based Development.

1. Component qualification. It is identified by characteristics (performance, reliability and usability) in their interfaces , and also checks reusable components.
2. Component adaption is needed. Because components will very less integrate with the system immediately.
3. Component composition, which integrates the components into a working system.

Components might change to be updated based on the requirements of system changes. In order to make sure updates to the components adhere to the system architecture being developed. Summarily, they must be qualified and adapted if reusable components are available for potential integration. They also must be engineered if new components are added. [18]

2.4 Aspect Oriented Programming

According to Kiczales et al. [10], traditional modularity approach cannot be applied in the construction of complex software systems. This is because they include aspects which cut across both one another and the actual executable code. These aspects include for example distribution and failure handling. The cross-cutting of aspects means that the software modules eventually become tangled with aspects. This is considered one of the main sources of complexity in software systems. *Aspect-Oriented Programming* (AOP) is a programming paradigm which deals directly with aspects of concern rather than modules of software code. The purpose of AOP is to remove the tangling by making it possible to express aspects of concern and then to combine those aspects with one another and executable code using automating tools.

Also in AOP *abstraction* and *decomposition* are used to break down complex problem. However, instead of functional decomposition, AOP is based aspectual decomposition. This means that in Aspect-Oriented programming, the software artifacts do not remain contiguous in the executable program. Instead, the code resides in separate aspect descriptions which are spread over the actual executable program. [10]

This enables decoupling the aspect-oriented code and other functionality. Therefore, the details of the aspects can be modified without having to modify all software code that the aspects affect. The modularization of the concerns is realized in AOP languages trough the following basic elements [2]:

- *Join point model* which specifies the points in the code that the aspect enhancements are added in
- means of identifying join points
- means of specifying join point behavior
- methods for combining aspects and their specifications
- methods for attaching aspects to the program

Aspect orientation has also been applied to software development stages preceding programming, thus developing a *Aspect-Oriented Software Development* (AOSD) and *Aspect-Oriented Design* (AOD) disciplines. It is therefore necessary to observe software architecture from Aspect-Oriented angle. However, software architecture and aspect orientation have evolved separately as disciplines which poses problems in this process. Reasons for this are two-folded: firstly, it is difficult to handle different aspects in a consistent, similar way in the architecture. Secondly, the same aspects can require different treatment in different systems. . In principle, integrating aspect into the software architecture requires defining the join points of the components and the connections between different aspects and components at the architectural level. It can be said that with such methods, the simplicity of the design process is difficult to preserve. [13]

2.5 Reuse Based Development

The NATO Software Engineering Conference in 1968 can be considered as a starting point for the software engineering view in general [16]. The aim of the conference was to tackle the so-called software crisis: there was a need to build complex software systems in a controlled and cost-effective manner. From the beginning, the reuse of software components was considered to be one of the main concerns in software engineering. In short, software reuse means using existing software artifacts in the construction of a new software system [11]. However, although the field in general has matured, software reuse has not become a standard practice. In addition to reusable pieces of code, software reuse includes reusing e.g. modules, specifications and documentation [5]. All software reuse techniques can be analyzed based on four different dimensions [11]:

1. *Abstraction*: it is essential that the reusable artifacts are abstracted so that the developers can easily determine whether a reusable artifact is feasible for their needs and how this artifact should be used in a specific context. Abstraction is the most important aspect of software reuse. The idea is that software typically consists of several layers of abstraction in which the higher layers hide the details of the lower layers. In other words, abstraction distinguishes specification representation on the higher level from the realization on the lower level.
2. *Selection*: artifacts are often categorized to guide the developers when they search the artifact library. Also the ways of retrieving and exposing artifacts are essential to any software reuse technique.
3. *Specialization*: software artifacts are often merged into generic artifacts. To be able to use them, the developers have to specialize the artifact trough parameters or other kind of refinement.
4. *Integration*: software developers construct complete software systems from reusable artifacts using an integration framework. An example of this kind of framework is module interconnection language in which functions are exported from modules that implement them and imported to modules that use them.

Due to the generic nature of these issues, a multitude of software reuse techniques can be recognized. Krueger et al. (1992) divide software reuse into the following eight categories:

– High level languages
– Design and code scavenging
– Source code components
– Software schemas
– Application generators
– Very high-level languages
– Transformational systems
– Software architectures

Accordingly, the difficulty in software reuse stems from the complexity of abstractions. The developers need to be familiar with the abstractions to be able to use the software artifacts. For example, if a library of mathematical operations is used, domain knowledge is required. The use of abstract data structures, in contrast, necessitates understanding of the underlying semantics. [11]

3 Comparing approaches

3.1 Scope of SOA vs. OO, CBD and AOP

The objectives of the four discussed software development approaches - Service-Oriented Architecture, Object-Orientation, Component-based Development and Aspect-Oriented Programming - are similar to some extent. They can all be considered as ways to promote software reuse and methods for structuring software systems into artifacts that can be managed separately for each other. However, these approaches have different scopes and focuses and they can be considered to operate on different levels of abstraction.

Conceptually, the approaches define different software system characteristics. By definition, services are software components. Clearly, a good service captures the basic characteristics of a component. However, the characteristics of SOA define in more detail the software architecture that these specific components constitute. For instance, services operate in distributed environment and focus on document-centric communication. In contrast, component-based development does not take that much stand on how the components interact with one another - this depends on the technology that the components are based on. On the other hand, components can provide the basis for services, i.e. service interfaces and the structure of exchanges messages is often based on the component specifications. In turn, components can encapsulate objects. Aspect-orientation, on the other hand, can be seen as a complementary paradigm affecting the software system on several levels. Aspects are closely connected to objects and their classes because they affect directly to the methods at the code level. However, aspects must be also taken into account when relationships between different components are defined. Similarly, they affect the composition of services and therefore the eventual structure of SOA.

Based on the above discussion, a layered organization of the different artifacts defined by the approaches (figure 1) can be outlined:

It can also be stated that the scope and level of ambition associated with each of the approaches is different. To truly leverage from SOA, one should aim at high service reusability at the enterprise level. Components provide similar benefits but on a smaller scale. Objects, on the other hand, can be adopted for a single application development project but they can still provide value. An enterprise-wide OO-architecture is of course also possible and significant, even if it would not result in the use of components or services. Finally, aspects add value especially if they are used across the enterprise. However, unlike services, they concentrate on a specialized, recurrent aspects, not reusable business functions like services.

3.2 Enterprise application with SOA, OO, CBD and AOP

SOA Software components in a SOA are services based on standard protocols, which are not just encapsulation of some code of the lower layer of application. A service is a software asset of distinctive functional meaning that encapsulates a

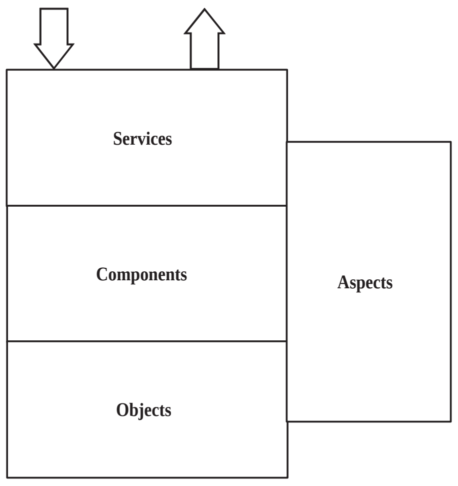

Fig. 1. A layered organization of Objects, Components, Services and Aspects

high-level business concept including contract, interfaces, implementation, business logic and data. OO and AOP have no corresponding elements for repositories. Object-Oriented (OO) programming is a programming paradigm which deals with relationship between objects, and Aspect-Oriented Programming is also a programming paradigm with deals directly with aspects of concern rather than modules of software code.

In SOA, Service repository provides facilities to discover services and acquires all information needed to the use of the service, especially if service must be discovered outside scope of the of enterprise; it also provides additional information to service contract like location, availability, QoS, Provider information, constraints and so on. Moreover, it is necessary to enable long term benefits and reuse.

Service Bus is used to connect all participants of an SOA. "In computing, an enterprise service bus (ESB) provides foundational services for more complex architectures via an event-driven and standards-based messaging engine, and generally provides an abstraction layer on top of an implementation of an enterprise messaging system which allows integration architects to exploit the value of messaging without writing code. ESB does not implement a SOA but provides the features with which one may be implemented. Although a common belief, ESB is not necessarily web-services based. Most ESB providers now build ESBs to incorporate SOA principles and increase their sales, e.g. Business Process Execution Language (BPEL)." [22]

CBD with COTS Modern enterprise application systems developing process become more and more large-scaled, uneasily controlled, complex. Also, due to time-to-market , no developing standard pressure and growing demand of searching for a cost-effective, efficient and satisfying multiple Quality of service (QoS) requirement software developing paradigm, enterprise application are developed by using commercial-off-the-shelf (COTS) components rapidly. Comparison to the traditional approach in which software systems can only be implemented from scratch; these COTS components can be developed by different vendor using different languages and different computer platforms. In CBD, COTS components can be checked out from a component repository, and assembled into a target software system. So that Enterprise applications increasingly developed using COTS component middleware. Component middleware encapsulates sets of services in order to provide reusable building blocks that can be used to develop enterprise applications more rapidly and robustly than those built entirely from scratch.[7] There are many examples of CTOS component middleware like the Common Object Request Broker Architecture (CORBA), Component Object Model (COM), Distributed COM(DCOM), Sun Microsystem's JavaBeans and Enterprise JavaBeans (J2EE), and emerging Web Services middleware such as .Net and ONE, based on XML and Simple Object Access Protocol (SOAP).

Currently, the growing importance of open source enabled services in the economy and in most areas of business has been widely recognized. It is a very economy way to do deployment of an SOA that integrates various OSS systems with CBD in order to reduce the cost and improve the work efficiency.

3.3 Reuse

AOP, OO, CBD and SOA do work at different levels of abstraction (3.1) . AOP at code level, OO at object level, CBD at component level and SOA at service level. This means that also reuse techniques are also working on different levels of abstraction. Software reuse can be divided to four different dimensions: abstraction, selection, integration and specialization (2.5, [11, p. 3]). To analyse differences in reuse techniques, these techniques can be categorized along dimensions (Table 1: Dimensions of reuse).

As it can be seen, SOA methods for reuse are quite similar to OO and CBD methods. Service Interfaces are equivalent of class or component interfaces from reuse perspective as data ownership is equivalent of data hiding and encapsulation.

Discoverability and use of service repositories is basic SOA principles to enable reuse. For CBD and OO comparable method can be use of class and component libraries, though their use is not as strongly tied to methodology as use of repositories in SOA. For AOP there is not equivalent selection strategy.

For integrating existing components, SOA has a concept of service compositions which is facilitated by service bus. Equivalent CBD and OO-methods are method or function calls and application framework or middleware such as object request brokers, message passing middleware and application servers. For AOP the joint point model allows integration of different aspects to complete application. Comparing SOA and OO strategies, service composition using service bus is technically close to RPC calls between objects or components.

Method / Dimension	CBD	OO	AOP	SOA
Abstraction	Component Interfaces, Encapsulation	Classes, Interfaces, Data hiding and encapsulation	Decoupling of aspects	Service interface/contract, Data ownership
Selection	Component libraries	Class libraries	NA	Repositories
Integration	Function class, ORBs etc.	Method calls, ORBs etc.	Join point model	Service composition, ESB
Specialization	NA	Inheritance	NA	Payload semantics

Table 1. Dimensions of reuse

For specialization SOA principle of payload semantics allows building more specialized components on existing code by adding new message fields. In object oriented programming inheritance is the method for creating sub-classes that can inherit implementation and interfaces from super-classes. These methods are not equivalent as payload semantics in itself only allows services to be extended but inheritance mechanism is built for the reuse in mind. Payload semantics could also be used in CBD or in OO development, but interface semantics and strong typing is usually used as it is built in for OO-languages. As CBD could be done in purely procedural language, inheritance has not been included as a reuse method to the table. The methods for reuse are not mutually exclusive. As SOA and OO have different levels of abstraction, SOA development can also use OO based reuse mechanisms if services are programmed using object oriented principles. AOP can be used on the same project to enable more reuse by separating aspect specific code from domain specific code if the selected framework and tools have support for it.

3.4 Layering and architectural patterns

Layering is one key architectural principle in traditional software development. Martin Fowler[4] suggests three key architectural layers that are used in enterprise applications (Table 2). Note: These layers are logical layers in application and can be distributed differently in normal N-tier architecture between tiers. For example in case of a thin client, most of the presentation logic could be on the server and in some cases to have faster response times some of the domain logic could be implemented on the client.

When compared to N-tier architecture, in SOA services can call each other without tiered layering approach. But still these three principal layers are represented in SOA application. Application front end is part of the presentation layer, domain logic is implemented in services and ESB integrates services. Same

Layer	Responsibilities
Presentation	Provision of services, display of information
Domain	Logic that is the real point of the system.
Data Source	Communication with databases, messaging systems, transaction managers, other packages

Table 2. Three Principal Layers

kind of architectural patterns as used in OO- or CBD-based development can be used in development of SOA services. According to Michael Stal [19] same patterns that have been found usable in J2EE applications are directly applicable to SOA. In figure 2 is a collection of J2EE patterns that Stal suggest for SOA development to minimize client-service communication, decouple clients and services from structural issues (such as service discovery) and increase developer productivity.

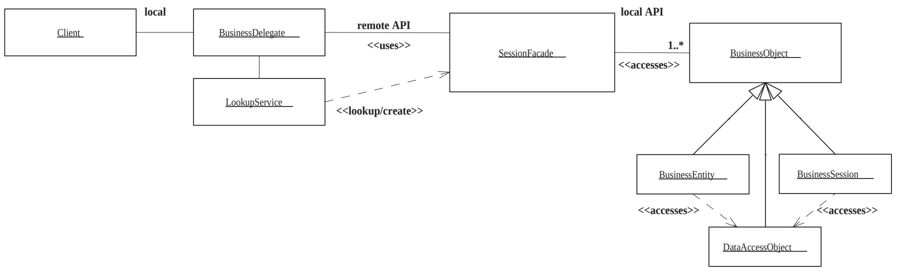

Fig. 2. Applying J2EE patterns in SOA-based context

Here *Business Delegate* shields client from communication details such as service discovery and message handling, *Session Facade* implements stateless and coarse-grained entities (services) and *Data Access Object* shields services from database and enterprise information systems.

3.5 Extending application

Regardless of the adopted approach, the developed software artifacts will change and evolve. Some changes can be made purely on the implementation level without changing the interface behavior of services, objects or components. These kinds of changes do not necessarily pose problems in the software system because this does not change the way the element is used externally. Also in case of AOP the aim is that the content of the aspect can be changed without affecting the main code. However, if the interface is changed by e.g. adding new operations or changing the parameters of the existing services, objects or components, a mechanism for version management is required. This is because the old version of the interface needs to be supported as well: no assumptions can be made about the use context of the software artifact.

The version and evolution management mechanisms of different component-based technologies are discussed by Stuckenholz (2005). Also Web Services are included. In short, it can be said that most of the component-based technologies provide basic mechanisms for distinguishing between different versions of the component. However, this might also require inflexible approaches such as manual assignment of version numbers. For example CORBA does not include any kind of support to component evolution. J2EE application servers use alternative class loaders to support applications using different versions of classes. .Net resources, in turn, packaged in to assembles which include a manually maintained manifest that covers all the version information related to the application. In the context of Web Services, rudimentary version management can be adopted using different namespaces in the Web Service Description Language (WSDL) files [1]. This way it is possible to run several versions of the same service in order to support different kinds of clients. Still it has been said that the Web Service architecture does not currently support versioning because the service provider cannot convey the changes to the requester (Papazoglou et al. 2006). More sophisticated mechanisms such as suitability checks or incompatibility warnings cannot be found in any of the component-based industry solutions [20]. The stronger the dependencies between components, the harder it is to manage different versions of them [20]. In other words, for all of the approaches, the version management problem can be partially avoided by following the core design principles they imply.

Applications can also be extended using technology-dependent extensions that specialize in a specific issue in the application such as security or quality of service. In case of Web Services, several proposals of such standards have been made including e.g. WS-Security and WS-Policy. These views have also been integrated into Extended Service-Oriented Architecture, xSOA [15]. However, it can be said that the extensions have not been widely adopted yet and the xSOA vision has not yet been realized. Extensions for component-based approaches depend on the used technology and their detailed discussion is out of the scope of this study. For example CORBA specification has been extended for several purposes. The quality standardization status of these is unclear, however. For example, CORBAsec has been considered inadequate due to its limited access right mechanisms [9]. The need for extensions depends on the adequateness of the original technology. Additionally, the layering of services, components and objects is meaningful here: for example, several security APIs exist for Java (objects) and they can be directly used in J2EE components. However, also the level of abstraction should be taken into account. For instance, only on the service layer can a composition language standard be considered relevant.

3.6 More concerns

Interoperability always plays an important key role in the enterprise application development under different platforms. Let us present the case of Web Service in order to clarify the importance of interoperability. Based on different developing approaches of Web Service, the lack of seamless interoperation can be found in different attempts to provide mappings between Web services abstracts and abstractions given by different middleware components. [26] So that, we need an interface design method based on identifying elementary business function and converting standard message formats (document) into a set of corresponding service interface in Web services according to SOA.

In order to satisfy the requirements of e-business application in the sense of interoperability, we can refine the message transformation relied on service interfaces design in service bus. Service-centric model provide a superior interoperability solution in comparison to document-centric model. Even though the main advantage of the document-centric approach is able to interoperate across different environments, as documents can be transmitted as message payloads with using a variety of messaging protocol. [6] But externalizing data structures in the form of document schemas creates dependencies between partners applications that make the document-centric approaching inflexible and difficult to evolve as changes in documentation specifications [6]. In service-centric model, Web services remove the need to use data interchange as interoperability mechanism for e-business applications and make service interfaces designed significantly in order to reduce coupling between applications with improving scalability. Service interfaces should clearly articulate the business operations they perform as well as the required input parameters, possible errors or exceptions, and results. Service interfaces should be easily understood by business experts who do not necessarily possess in-depth technical skills. This allows business experts to use the services productively to compose business processes and applications. [8]

To refine interface design, 1) input parameter should form a minimal set. 2) Output parameter should also form a minimal set. 3) Output parameter must be fully functionally dependent on the input parameter set. [6] Those three interface design rules are to minimize inter-dependencies between applications.

Moreover, as we discussed in some previous sections, the SOA approaching comes up certain requirements on top of OO and CBD methods. It is not a good idea and approach to apply directly the existing OO and CBD concepts for the purpose of modeling the SOA. The OO concept is still mainly implementation-related for SOA, while CBD methods are mainly focus on finer-grained components. According to the business-driven character of SOA, a proper developing approach is to combine CBD and OO concepts on activities and work flow.[25]

4 Discussion

It can be said that SOA itself it not a technical novelty. Technical SOA principles like data ownership are basically object oriented principles. As SOA is technically agnostic, any messaging techniques can be used in communication - in this respect SOA services can be seen as normal software components communicating over message passing middleware. What separates SOA services from common components is the requirement of business functionality which raises the level of abstraction.

As stated in chapter 3.1 SOA functions in higher level of abstraction than other techniques. Still the purpose of abstraction is same as in OO- and CBD-methods, make domain logic more understandable by raising the abstraction level. AOP on the other hand does not raise the level of abstraction, but rather is separating different concerns like security from the domain logic. It could be said that AOP is more independent from other techniques as its use is not as tightly tied in as SOA, CBD and OO are. It is easy to imagine a case where AOP is not part of the project tool set, but not a situation where SOA is implemented without using OO- or CBD-techniques.

As SOA services can be built on CBD- and OO-principles (see figure 1 and figure 2), it adds a new layer for reuse. In addition to reuse of software components and objects also full blown services can be reused via standard communication over ESB and discoverability offered by repositories. There is some evidence that SOA really increases reuse possibilities - one case is reported on eWeek [21], thought they also attribute increased reuse for use of high level languages.

SOA applications by nature are distributed, which means their construction is hard. XML- and Web Services may make the communication easier, but that is only one aspect of distributed application. There still are questions about security, transactions, fault tolerance, change management etc. which are hard problems. Steve Vinoskis complaint about new technologies only making easy things easier [24] is quite easy to understand in this perspective. Even if his article does not handle SOA itself - an SOA application will also have to have answers to these questions.

5 Future

Compared to the Service-Oriented approach, Object-Oriented and Component-Based paradigms have a relatively long history behind them. Therefore, solid methodology for developing Object-Oriented or Component-Based applications exists. As the SOA paradigm matures, there is a need for such efforts in this context as well. This requires careful consideration of the role of different software artifacts in the system: one should clearly distinguish between reusability on different levels, for instance. The MDA approach may provide a feasible framework for dealing with different levels of abstraction in software systems. MDA is closely related to the Object-Oriented approach but it can also be applied together with SOA. This clearly represents one future research direction. Furthermore, many non-functional issues such as security or reliability have not still been completely solved by these paradigms. This means that the supporting Web Service specifications need to be further developed and evaluated and their relationship to Component and Object level defined.

References

1. Brown, K., Ellis, M.: Best practice for web service versioning - keep your web services current with wsdl and uddi. Technical report, IBM (2004)
2. Elrad, T., Aksit, M., Kiczales, G., Lieberherr, K., Ossher, H.: Discussing aspects of aop. Commun. ACM **44**(10) (2001) 33–38
3. Erl, T.: Service-Oriented Architecture : Concepts, Technology, and Design. Prentice Hall PTR (August 2005)
4. Fowler, M.: Patterns of Enterprise Application Architecture. Addison-Wesley (2003)
5. Freeman, P.: Reusable software engineering: concepts and research directions. In: ITT Proceedings o/ the Workshop on Reusability in Programming, pages 129- 137. (1983)
6. George Feuerlicht, S.M.: Design method for interoperable web services, ACM, Sydney, NSW (2007)
7. Gokhale, A., Schmidt, D.C., Natarajan, B., Wang, N.: Applying model-integrated computing to component middleware and enterprise application. Communication of the ACM (Octorber 2002) 65
8. Hanson, J.: Coarse-grained interfaces enable service composition in soa url: http://builder.com.com/5100-63865064520.html (accessed 14.05.2007). Web (August 2003)
9. Hauf, M., Schwarz, J., Polze, A.: Role-based security for configurable distributed control systems. words **00** (2001) 111
10. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In Akşit, M., Matsuoka, S., eds.: ECOOP '97 Object-Oriented Programming 11th European. Volume 1241 of Lecture Notes in Computer Science. Springer-Verlag, New York, NY (June 1997) 220–242
11. Krueger, C.W.: Software reuse. ACM Comput. Surv. **24**(2) (1992) 131–183
12. Meservy, T., Fenstermacher, K.: Transforming software development: an mda road map. Computer **38**(9) (2005) 52–58
13. Navasa, A., Prez, M., Murillo, J., Hernndez, J.: Aspect oriented software architecture: a structural perspective (2002)
14. Papazoglou, M.P., Georgakopoulos, D.: Service-oriented computing. Commun. ACM **46**(10) (2003) 24–28
15. Papazoglou, M.: Extending the service oriented architecture. Business Integration Journal (2005)
16. Randell, B.: Software engineering in 1968. In: ICSE '79: Proceedings of the 4th international conference on Software engineering, Piscataway, NJ, USA, IEEE Press (1979) 1–10
17. Sethi, R.: Programming Languages 2ed. Addison-Wesley (1997)
18. Siddiqui, F.: Component based software engineering, a look at reusable software components (August 2003)
19. Stal, M.: Using architectural patterns and blueprints for service-oriented architecture. Software, IEEE **23**(2) (2006) 54–61
20. Stuckenholz, A.: Component evolution and versioning state of the art. SIGSOFT Softw. Eng. Notes **30**(1) (2005) 7
21. Taft, D.K.: Usi finds soa key to reusability. eWeek **24**(1) (January 2007) D1–D4
22. Various: Enterprise service bus url: http://en.wikipedia.org/wiki/enterprise_service_bus (accessed 14.05.2007). Wiki (2007)

23. Various: Software componentry
url: http://en.wikipedia.org/wiki/software_componentry (accessed 14.05.2007). Wikipedia (2007)
24. Vinoski, S.: The more things change... IEEE Internet Computing **8** (2004) 87–89
25. Zirpins, C., Lamersdorf, W., Piccinelli, G., Finkelstein, A.: Object orientation and web services. (1 2005) 1–9
26. Zoran Stojanovic, Ajantha Dahanayake, H.S.: Agile modeling and design of service-oriented component architecture. (June 2003)