Designing a Generic and Evolvable Software Architecture for Service Oriented Computing

Herwig Mannaert, Kris Ven and Jan Verelst

University of Antwerp, Department of Management Information Systems
Prinsstraat 13, B-2000 Antwerp, Belgium

Abstract. Service Oriented Architecture (SOA) is becoming the new paradigm for developing enterprise systems. We consider SOA to be concerned with high-level design of software, which is commonly called *software architecture*. In this respect, SOA can be considered to be a new architectural style. This paper proposes an advanced software architecture for information systems. It was developed by systematically applying solid software engineering principles such as *loose coupling*, *interface stability* and *asynchronous communication* to contemporary n-tier architectures for information systems in Java Enterprise Edition. The resulting architecture is SOA-compliant, generic and demonstrates to a high extent architectural qualities such as evolvability.

1 Introduction

In the last few years, Service Oriented Architecture (SOA) has been proposed as a new paradigm for building enterprise systems. Basically, the idea behind SOA suggests that systems should be built of services operating in highly networked environments. Since these services are modular and exhibit loose coupling, SOA should lead to evolvable systems. SOA is most often implemented by using Web Service technology. However, several authors emphasize that services can be composed of object oriented code, or even legacy code [1–3].

Building a SOA-compliant enterprise information systems for a specific organization is, however, not straightforward. From a technical point of view, one of the challenges is that SOA requires highly sophisticated designs to ensure that not only current, but also future requirements can be met. This means that the cost and effort in developing a full-scale SOA for a given organization is substantial, and for many organizations maybe even prohibitive. On the other hand, there seems to be a degree of similarity between the enterprise systems of most organizations. An indication of this is for example that most systems are based on a standard software package, with mostly limited customizations. This suggests that it may be possible to build an architecture for real-world, large-scale enterprise systems, which implements SOA-principles and can be used by a wide range of organizations.

In this paper, we propose an advanced, generic software architecture that could be used for building enterprise information systems. Initially, the architecture was developed for application domains such as large-scale satellite-based content distribution,

Mannaert H., Ven K. and Verelst J. (2007).
Designing a Generic and Evolvable Software Architecture for Service Oriented Computing.
In *Proceedings of the 1st International Workshop on Architectures, Concepts and Technologies for Service Oriented Computing*, pages 9-18
DOI: 10.5220/0001348600090018

monitoring and control of remote power units, and communications monitoring systems, but a prototype has shown its potential for building enterprise information systems. It was built according to contemporary n-tier architectures for information systems in the Java Enterprise Edition (Java EE) framework. The software architecture was built by systematically and thoroughly applying solid software engineering principles such as *loose coupling*, *interface stability*, and *asynchronous communication*. The resulting architecture is suitable for large-scale systems, generic and demonstrates to a high extent architectural qualities such as evolvability. The architecture is also SOA-compliant since it supports many SOA-principles, including loose coupling, reusability and abstraction [4]. The software architecture is independent from the underlying implementation technology (e.g., web services), but has been fully implemented in Java EE and is in use in several organizations.

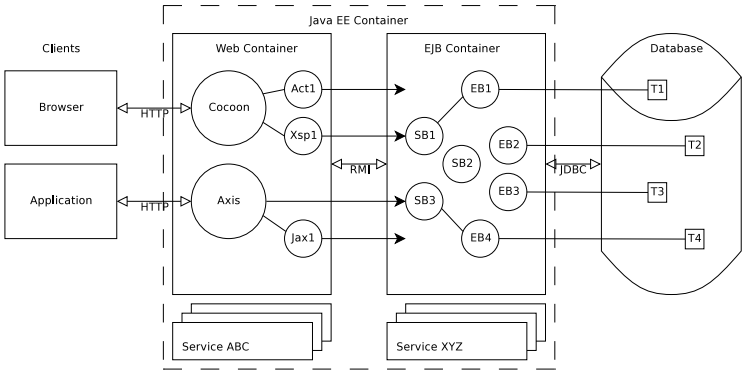
2 Software Architecture

SOA is a holistic concept spanning many research areas, from technical issues such as web services to management issues concerning business processes. However, our point of view is that SOA concerns essentially high-level design of software. This level is commonly called *software architecture*, and is a growing field of research within the area of software engineering [5]. More specifically, SOA can be seen as a new architectural style [6]. For example, Lublinsky considers SOA as an architectural style "*[. . .] promoting the concept of business-aligned enterprise services as the fundamental unit of designing, building, and composing enterprise business solutions. Multiple patterns, defining design, implementations, and deployment of the SOA solutions, complete this style.*" [7]. SOA attempts to increase modularity, thereby improving evolvability of the entire system. Also, considering SOA as a high-level design issue implies that SOA is more general than an implementation technology such as web services (i.e., web services is only one possible implementation technology for SOA).

Currently, client-server architecture [6, 8] is frequently used for developing information systems. Java EE, for example, is based on an n-tier client-server architecture. The software architecture we propose, is an attempt to SOA-enable these n-tier client-server architectures, or in other words, to extend them according to SOA-principles.

In order to visualize a software architecture, different views are necessary [9]. Each view differs in its intended stakeholders, and the system properties that are described. The *physical topology view* of the architecture that we propose is depicted in Fig. 1. Consistent with contemporary design principles, the concept of layering is adopted here. Each layer is highly cohesive, and loosely coupled to the other two layers. This principle ensures that modifications to a specific layer have no—or limited—impact on the rest of the system. This requires that each layer has an interface that shields the internal implementation details of that layer. This interface should remain stable in time (see Sect. 3.2). By introducing layers into the systems architecture, volatility can be better managed, as coding changes will not propagate across different layers.

Nowadays, information systems are generally composed of minimum 3 tiers: the *user interface tier*, the *business tier* and the *database tier*. In contrast to the traditional 3-tier design, we distinguish between 4 different tiers: the *client tier*, the *web tier* (con-



Legend: EB: Entity Bean; SB: Session Bean; Jax: JAX-RPC; Act: Cocoon Action Object; Xsp: XML Server Pages; T: Table

Fig. 1. 4-Tier Application Architecture.

taining for example Cocoon and Axis), the *EJB tier* and the *database tier*. Both the web and EJB tier are grouped in the Java EE container.

3 Guiding Principles

The architecture is based on several solid software engineering principles such as *loose coupling*, *interface stability*, and *asynchronous communication*. These principles are already known for quite some time. However, our main contribution consists of applying these principles in a systematic and thorough way. This allowed us to improve upon several architectural qualities such as evolvability, performance, security and availability [10]. In this paper, our focus will be on the evolvability of the architecture.

3.1 Loose Coupling

Loose coupling is an important principle in software engineering that aims to minimize the degree of interconnections (or coupling) between modules. If a module has a large number of connections to other modules, the module is also dependent on these other modules. As a result, the complexity of the system increases. We have applied the principle of loose coupling consistently throughout our architecture, by minimizing the number of interconnections between modules. In fact, we strive towards linking only two modules at the same time, in order to keep the complexity of the system to a strict minimum. We will provide several illustrations of this in the following sections.

3.2 Interface Stability

Evolvability is essential for an information system in order to accommodate changing requirements. In large-scale distributed systems, updating client applications following the release of a new version of a service provider is not always feasible. This new

version could incorporate additional features and/or additional interfaces that are accessible to clients. In this section, we only consider *extensions* in the interface (i.e., the addition of parameters). This should however not affect the ability of existing clients—that will not use this new functionality—to keep accessing the service provider. We refer to this principle as *version transparency*. Hence, it is necessary that the interface of the service provider remains stable in time. We distinguish between two types of interface stability.

A first type is *strict-sense interface stability*. This type of stability requires loose coupling between modules that is completely implementation technology independent (i.e., it does not require that the service provider is based on a specific implementation technology such as Java EE). Consequently, the use of XML (e.g., web services that communicate via SOAP) is mandatory for the exchange of information between modules. This type of loose coupling is situated at run-time level, as recompilation of client applications is not required when a new version of a service provider is released. This type of loose coupling is preferable when there is a large number of distributed clients, or when the service client is located in a different unit of compilation than the service provider.

A second type of interface stability is *wide-sense interface stability*. This type respects the principle of loose coupling by only passing serializable objects with default constructors that only provide access to member fields through get and set methods. However, it allows imposing the use of a specific implementation technology (e.g., Java RMI). This type of coupling is situated at compile-time, since it requires recompilation of client applications upon the release of a new version of a service provider. However, coding changes to the service provider do not propagate beyond the service interface, i.e., modifications to a service provider should not require any coding changes to existing clients. Wide-sense interface stability can be a valid option when the number of clients is limited, or when clients are contained within the same unit of compilation as the service provider.

3.3 Asynchronous Communication

In general, service invocations tend to be synchronous: the client requests an operation from a service provider, waits for the provider to complete its operation, and receives the result of this operation. This is for example how web services essentially work. Synchronous communication however has some serious drawbacks.

First of all, the use of synchronous communication creates *temporal* coupling between modules [7]. This means that the client is blocked from the time that it issues the call until it receives a reply from the service provider. This may have negative performance consequences. It also requires that the service provider is available at the time the client issues the service request (i.e., the provider system is up and running, and there is network connectivity between service client and provider). Second, synchronous communication does not allow for the state of the transaction to be known. It is for example not straightforward to determine whether a service request has been submitted, but has not arrived yet at the service provider. Finally, when using synchronous communication, the client must incorporate additional knowledge about the underlying layers in

the information system. This once again increases coupling, and as a result, the complexity of the system increases. For example, if a client has a user interface that retrieves data from a service provider, the user interface has to respond to the possibility that no network connection could be established to the service provider. However, it must also be able to react upon other errors that occur on the provider side, e.g., the fact that the database is currently down.

4 Architectural Patterns

We argue that additional structure is required, on top of standard component models and frameworks such as Java EE, Cocoon and Axis (see Fig. 1), in order to support the principles that were discussed in Sect. 3. Therefore, we have developed four different *architectural patterns* that are based on elementary object types, namely: *data objects*, *flow objects*, *action objects* and *connector objects*. Each of these patterns is cross-layer, since each pattern defines a number of objects located in several layers in Fig. 1. Although we do not claim that these patterns are the best possible solution, we have found them to be suitable for describing changes in a quantitative way [11], as well as automatic code generation [12]. It is important to note that these patterns are not solutions which are tied to a specific implementation platform. Instead, they are based on fundamental software engineering principles and concepts. In this paper, we illustrate how these underlying principles and concepts can be implemented in a certain technology (e.g., Java EE).

A code base has been developed within the Java EE framework. JOnAS is used as application server, while the Cocoon XML publishing framework provides the user interface. The code base consists of about 1200 Java classes, containing about 120 EJBs divided over 8 separate software components, and provides 5 different applications. These applications are divided in three different application domains: satellite-based content distribution [13], monitoring and control of remote power units [14], and communications monitoring systems (i.e., nurse call systems in hospitals and digital processing in a broadcast studio). Three components are shared by all five applications, the other components are currently confined to a single application. Given the genericity of this architecture (which is based upon the four architectural patterns), we are convinced that the architecture can be used to build enterprise information systems.

In order to build applications within the architecture, the universe of discourse (i.e., the relevant part of the real world) is modeled in terms of these four architectural patterns. For example, to develop an application for an on-line book store, data objects can be used to contain information on the books in the catalogue, connector objects can be used to generate sales reports and to provide the user interface, flow objects can be used to handle a sale, and action objects can be used to register payment through a credit card. We will now describe each of these patterns in more detail.

4.1 Data Objects

Data objects represent persistent objects in the real world that are stored in a relational database. Examples of data objects are *customer* and *order*.

Within the Java EE framework, data objects are implemented by using entity beans. For each object `<Obj>` that is stored persistently in the database, the Java EE framework requires the implementation class (`<Obj>Bean`), the interfaces for the lifecycle operations (find, create, and delete) (`<Obj>HomeLocal` and `<Obj>HomeRemote`), and the interfaces for the business methods (`<Obj>Local` and `<Obj>Remote`).

In addition to these five standard classes and interfaces, we include two additional *transport objects* for each persistent object. Transport objects are serializable objects that encapsulate data fields of corresponding data objects and only provide getters and setters to access each field. They also have a default constructor (without parameters), in which default values are set for all its member fields. A first transport object (`<Obj>Details`) contains all data fields of an entity. A second transport object (`<Obj>Info`) contains a subset of these data fields. The idea here is to include only those fields in the info-object that will be shown in for example listings and tables that display summary information.

These transport objects are essential to the architecture, since they support the principles of interface stability and version transparency. As a rule, only transport objects are allowed as parameters or return value in the interface of service providers. This allows for loose coupling—and version transparency—between service client and service provider. Our architecture supports both *wide-sense* and *strict-sense* interface stability. Wide-sense interface stability is implemented by exchanging serialized transport objects with remote session beans by using Java RMI calls. This design ensures that recompiling the client is sufficient when the service provider is extended in functionality through the addition of parameters (i.e., no coding changes to existing clients are required). Strict-sense interface stability is obtained by serializing the transport objects to XML format, and invoking the web service that corresponds to the session bean at the service provider[1].

Moreover, our architecture supports dynamically changing (i.e., at run-time) how clients will call a service provider interface. The client will either call the session bean over Java RMI, or the corresponding web service by using XML messages. How the client must invoke the remote service is stored in the database at the service provider side. This setting can be changed at run-time. This means that—theoretically—the degree of coupling between client and provider can be changed as well. However, given the fact that the client must be able to support Java RMI in this situation, it means that the client must be recompiled when the service provider is modified (unless the client will only use web service calls in the future). This means that such clients are not strict-sense version transparent. This feature however provides opportunities for future evolvability of the system, and to make decisions on the architectural qualities at run-time. For example, if one initially wants to maximize performance, service invocations can take place over Java RMI. If, at a later time, the network configuration changes and a firewall is placed between the client and the service provider, the client can be reconfigured to invoke the corresponding web service.

[1] Within Java EE, session beans can be made available as web services.

4.2 Connector Objects

A connector object is used to import and export data objects from and to the outside world. Connectors can be used to transform data objects from and to: the *user interface* (e.g., HTML), *files* (e.g., PDF), and *network protocols* (e.g., UDP, HTTP, SNMP). Connector objects for example generate an entry form for an entity in the database, or generate a report in PDF format.

Within the Java EE framework, a session bean (`<Obj>ConnectorBean`) is created for each data object that will be imported or exported. This bean depends on at least one implementation class for a specific protocol (`<Obj><Protocol>`). This class is not an EJB and is independent from Java EE. The `<Protocol>` is a variable that allows for alternative implementations for a specific connector (e.g., to provide multiple implementations for sending and receiving network packets over TCP or UDP). This supports the dynamic configuration of different protocols or formats through dynamic class loading.

This design further builds on loose coupling. By dividing the responsibility of the import/export functionality between the session bean and the implementation class, the complexity of the system is kept to a minimum. The connector object (i.e., session bean) is part of the EJB framework, and has knowledge about the data model of the corresponding data object. It has however no knowledge about the specific implementation of the external format or protocol. The latter responsibility is assigned to the implementation class, which however has no knowledge about the EJB framework. The same principle is used at the user interface. The Cocoon action classes for example have knowledge of Cocoon and the data model, but not about the underlying EJB container. As such, each object in the system only has knowledge about (is coupled with) maximum two other objects, hence minimizing complexity.

4.3 Flow Objects

Flow objects represent business processes, i.e., a sequence of steps in a workflow. Examples of flow objects are objects that handle the processing of a new order, or the registration of a new customer. In our architecture, a workflow is considered to be a sequence of actions (implemented by action objects, see Sect. 4.4).

Within Java EE, a flow object is an entity bean (`<Flow>OrderBean`) that stores the consecutive transitions required to execute a workflow. This bean also captures the current state of the workflow. Each transition is stored as a persistent object by using another entity bean (`TransitionBean`). This entity bean contains information such as the input and output state, and a reference to the session bean (i.e., action object) that implements the transition. The editing of the workflow is supported by the entity bean (`<Flow>OrderBean`) which provides CRUD (create, read, update, delete) functionality.

An important advantage of storing workflow persistently in a relational database, is that it allows for dynamic reconfiguration. Within our architecture, it is possible to update the workflow within the application through a web-based interface. Although the Business Process Execution Language (BPEL) is often used to describe workflow, the disadvantage of BPEL is that it doesn't directly support persistency, nor concurrent

access with transactional integrity. More particularly, editing a BPEL file in XML format through a web-based interface is not trivial. However, in order to support BPEL specifications, it is possible to develop connector objects to import a BPEL file, parse the XML, and store its contents in a relational database.

4.4 Action Objects

Action objects are atomic steps in a workflow. Action objects perform operations on data objects, or external resources such as files. Examples of actions are encrypting a file, and performing a credit card validation.

In Java EE, action objects are implemented by using session beans. Similar to data objects, action objects require an implementation class (`<Act>Bean`), the lifecycle interfaces (`<Act>HomeLocal` and `<Act>HomeRemote`) and the business method interfaces (`<Act>Local` and `<Act>Remote`).

Similar to connector objects, we applied the loose coupling principle. This means that the action itself is implemented in a separate class (`<Act><Implementation>`), which has no knowledge of the Java EE framework. In order to increase flexibility and ensure loose coupling, `<Implementation>` is a variable that allows for providing several alternative implementations for a specific action (e.g., to support payments via various credit card companies using different interfaces). Since `<Implementation>` is a variable, it allows to dynamically choose between various implementations at run-time.

Some actions need to be performed regularly (e.g., every hour). For such actions, an EJB session bean (`<Act>EngineBean`) and an EJB entity bean (`<Act>ServiceEngineBean`) are created. The latter represents a persistent object that controls the time interval at which the action needs to be run, and also allows to start and stop the action. If the target of the operation is a persistent object that is represented by an entity bean (i.e., a data object `<Obj>Bean`), the state of the action can also be stored persistently as an entity bean (`<Obj>TaskState`).

The implementation of workflow through flow and action objects is fully based on asynchronous communication. This ensures loose coupling between service client and provider. As a result, different action objects implementing consecutive steps in a workflow do not communicate directly with each other. Instead, the input for a given action is stored in a database table. Each action has an agent that regularly polls the database table for incoming requests. When an outstanding request is found, the corresponding action is performed on the data. The output of this action is written to a second table in the relational database. The client (i.e., flow object) that has requested the action will also regularly poll the database for the result of the action. Once the result is available, it will be retrieved. This output can be passed as input to the next step in the workflow. It is clear that this design functionally and temporally decouples consecutive steps in a workflow. Another advantage of this design is that all actions and intermediate results of actions are logged in the database, and can be retrieved at any time. This allows for the creation of test data based on real operations that were performed by the system in the past, rather than artificially created data. This historic information may also be used for audit purposes.

5 Conclusion

In this paper, we have presented an advanced software architecture for information systems. The architecture is consistent with contemporary n-tier architectures, and demonstrates to a high extent several architectural qualities. The architecture has several important contributions.

First, the software architecture is generic, which is supported by several properties. For example, the application framework developed within this software architecture provides five different applications in distinct application domains. The architecture is also independent on the implementation technology (we have chosen to implement the architecture in Java EE). Additionally, it is possible to dynamically reconfigure several properties of the system at run-time, by using CRUD operations on the system itself. Examples are the degree of coupling between modules, and the workflows contained in the system.

Second, we have developed four different architectural patterns that are used as elementary building blocks within the architecture. This implies that the patterns can also be used for implementing meta-activities that represent common operations on services (such as discovery, selection and monitoring). These patterns are independent from a specific implementation platform, and are based on several solid software engineering principles such as loose coupling, interface stability, and asynchronous communication. By thoroughly and systematically applying each of these principles, we considerably increased several quality factors such as evolvability. This is illustrated by various characteristics of the architectural patterns. Transport objects (part of the data object pattern) support the notion of interface stability and version transparancy. This allows to extend their interface without requiring a recompilation of existing clients. Moreover, the architecture allows to choose at run-time between service invocations over Java RMI or web services, allowing for example to cope with changing requirements in the network infrastructure. Both the connector and action objects support the concept of dynamic class loading. This allows to provide additional implementations where clients can choose from. These patterns also support loose coupling and asynchronous communication, thereby separating the implementation as much as possible from the rest of the platform. The flow objects support run-time modifications to the workflow that is stored persistently in a relational database. This allows to update the workflow without any recompilation.

Third, the architecture is SOA-compliant, in the sense that it implements the aforementioned software engineering principles which also constitute the core of SOA, irrespective of the underlying implementation technology (e.g., web services).

Our goal is to further validate and extend this architecture in several ways. First, although we have implemented and tested the architecture in a number of settings, we plan to develop additional applications in other application domains. More specifically, we are convinced that this architecture is appropriate for building enterprise information systems, and will build on the current prototype to demonstrate this in more detail. Second, research can be performed on how the real world can be mapped to the four architectural patterns. Finally, we aim to identify additional patterns across the four architectural patterns that allow for the automatic generation of fully working code, called *pattern expansion*. A first pattern that was successfully expanded is the CRUDS

pattern, which involves the generation of classes that implement data and connector objects, and is described in previous work [12]. In order to develop information systems in this architecture, the developer needs to define the actions and the data model of the application. Based on these elements, a considerable portion of the source code can be automatically generated through pattern expansion.

References

1. Zimmermann, O., Krogdahl, P., Gee, C.: Elements of service-oriented analysis and design (2004) IBM Developerworks, on-line available at `http://www-106.ibm.com/developerworks/library/ws-soad1/`.
2. Papazoglou, M.P., Traverso, P., Dustdar, S., Leymann, F., Krämer, B.J.: Service-oriented computing: A research roadmap. In Cubera, F., Krämer, B.J., Papazoglou, M.P., eds.: Service Oriented Computing (SOC). Number 05462 in Dagstuhl Seminar Proceedings, Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany (2006)
3. Marks, E.A., Bell, M.: Service-Oriented Architecture: A Planning and Implementation Guide for Business and Technology. John Wiley and Sons, Inc., Hoboken, NJ, USA (2006)
4. Erl, T.: Service-Oriented Architecture: Concepts, Technology, and Design. Prentice Hall PTR, Upper Saddle River, NJ, USA (2005)
5. Shaw, M., Clements, P.: The golden age of software architecture. IEEE Software **23** (2006) 31–39
6. Shaw, M., Garlan, D.: Software Architecture—Perspectives on an Emerging Discipline. Prentice Hall, Upper Saddle River, NJ, USA (1996)
7. Lublinsky, B.: Defining SOA as an architectural style (2007) on-line available at: `http://www-128.ibm.com/developerworks/library/ar-soastyle/index.html`.
8. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. Addison-Wesley, Reading, MA, USA (1998)
9. Kruchten, P.: The 4+1 view model of architecture. IEEE Software **12** (1995) 42–50
10. Kazman, R., Klein, M., Barbacci, M., Longstaff, T., Lipson, H., Carriere, J.: The architecture tradeoff analysis method. In: Proceedings of the Fourth IEEE International Conference on Engineering Complex Computer Systems (ICECCS'98). (1998)
11. Mannaert, H., Verelst, J., Ven, K.: Towards rules and laws for software factories and evolvability: A case-driven approach. In: Proceedings of the International Conference on Software Engineering Advances (ICSEA'06), Tahiti, French Polynesia, October 29–November 3. (2006)
12. Mannaert, H., Verelst, J., Ven, K.: Exploring concepts for deterministic software engineering: Service interfaces, pattern expansion and stability. In: Proceedings of the Second International Conference on Software Engineering Advances (ICSEA 2007), Cap Esterel, French Riviera, France, August 25–31. (2007)
13. Mannaert, H., De Gruyter, B., Adriaenssens, P.: Web portal for multicast delivery management. Internet Research **13** (2003) 94–99
14. Mannaert, H., Huysmans, P., Adriaenssens, P.: Connecting industrial controller to the internet through software composition in web application servers. In: International Conference on Internet and Web Based Applications and Services, Mauritius, May 13–19. (2007)