Please quote as: Mauro, C.; Leimeister, J. M. & Krcmar, H. (2010): Service oriented device integration - An analysis of SOA design patterns. In: Hawaii International Conference on System Sciences (HICSS) 2010, Kauai, USA

# Service Oriented Device Integration – An Analysis of SOA Design Patterns

Christian Mauro[1] Jan Marco Leimeister[2] Helmut Krcmar[1]

[1]Technische Universitaet Muenchen
Information Systems
Boltzmannstrasse 3
85748 Garching, Germany
{mauro, krcmar}@in.tum.de

[2]Universitaet Kassel
Information Systems
Nora-Platiel-Strasse 4
34127 Kassel, Germany
leimeister@uni-kassel.de

## Abstract

*Service oriented device architecture (SODA) is a promising approach for overcoming interoperability issues and especially for extending the IT support of business processes to devices. It is based on the encapsulation of devices as services, and therefore on design principles of service oriented architectures (SOA). However, there is a lack of generalized concepts that resolve SODA-specific design problems. This paper contributes to this research gap by a) identifying a set of SODA-specific design problems, b) analyzing existing SOA design patterns regarding their applicability for SODA, and c) proposing a set of new pattern candidates which resolve open SODA design problems.*

## 1. Introduction

Service oriented device integration (also known as SODA – Service Oriented Device Architecture) is a promising approach to overcome interoperability issues, especially for extending the IT (Information Technology) support of business processes to devices [16] and therefore for a better alignment between IT and business (cf. [18]).

By performing a literature analysis, we identified 26 relevant papers in the field of SODA [20]. Even if these works achieved fundamental results by demonstrating the technical feasibility as well as the benefits of the concept, much room for further research was identified by our analysis. The research gap we address in this paper is the lack of generalized design concepts for SODA. Presented architectures in existing works are based on the specific requirements of the given scenarios (e.g., [23]) or are too abstract for resolving general SODA design problems (e.g., [5]).

The SODA concept is based on SOA (Service Oriented Architecture) principles. Several SOA best practices exist, concentrated in the form of SOA design patterns [9]. From an integration point of view (as we show in section 3), devices can be considered as software systems with specific hardware characteristics. Due to the fact that existing SOA design patterns address software systems only, the following procedure for identifying general SODA design concepts is pursued in this paper:

- *Step 1*: Analysis of the characteristics of devices in comparison to software systems.
- *Step 2*: Identification of specific SODA design problems, based on step 1.
- *Step 3:* Analysis of existing SOA design patterns, regarding their applicability for SODA.
- *Step 4*: Proposal of new SOA design pattern candidates that address unsolved SODA design problems.

This paper contributes to the SODA research field in three ways by delivering the following outcomes:

a) A set of identified SODA design problems.
b) A set of existing SOA design patterns which contribute to SODA or even resolve SODA design problems.
c) A set of new SOA design pattern candidates for the purpose of resolving open SODA design problems.

The structure of this paper is based on the presented procedure, each step having its own section. In addition, the general concept of SODA as well as the concept of patterns is presented in section 2. Section 7 summarizes the results and concludes with suggestions for future research.

## 2. Fundamentals

### 2.1. Service oriented device integration

The basic idea behind the concept of SODA is the encapsulation of devices as services, analogous to enterprise services in service oriented architectures [5]. An enterprise service is a software component that offers a business functionality on a highly semantical

level by specifying the interface in a standardized way (e.g., by the Webservice Description Language – WSDL) [17]. Highly semantical level especially means, that a service is self-descriptive in a way that it can be consumed dynamically, and loosely coupled by other components with a consistent understanding of shared data.

As shown in Figure 1, three layers can be distinguished (following concepts of [17]):

- *Device Layer*: This layer contains physical devices.
- *Device Service Layer*: As services encapsulated devices are placed on this layer, the device services use physical devices of the device layer.
- *Composition Layer*: Services can be combined to fulfill more complex logic, up to process logic [7]. Thus, the composition layer uses services of the device service layer.

This layer classification is used in the following sections to categorize identified design problems.

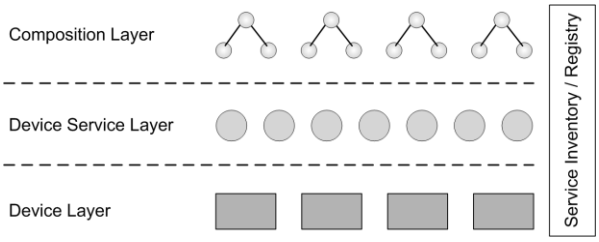


**Figure 1: Service oriented device integration**

### 2.2. SOA patterns

The idea of patterns can be traced back to Alexander [1] in the field of architecture and Gamma [12] in the area of software engineering. Basically, a pattern consists of the following elements [3] [22]: the *context* of a given problem and its circumstances, the description of the *problem* itself (also called forces), the proposed *solution* for the problem and *references* to related patterns.

Erl [9] extends this meta-definition of a pattern by using:

- a Pattern Profile (consisting of a requirement definition, an icon, a summary table, a problem definition, a solution, an application description, impacts, relationships and a case study example) and
- a Pattern Notation (including specific symbols to represent different kind of patterns as well as different types of pattern figures to emphasize specific aspects of the pattern).

Patterns "provide field-tested solutions to common design problems" in a "standardized and easily 'referencable' format" [9]. In addition, they are "generally repeatable" and "ensure consistency in how systems are designed and built." [9]. Thus, the application of patterns to realize the SODA concept is reasonable.

## 3. Characteristics of devices

Devices and software systems have common characteristics. Both are a piece of software running on a hardware platform. In addition, both are equipped with interfaces for accessing specific functionality (devices or software systems without such interfaces are not relevant for this paper).

On the other hand, there is a major difference between devices and software systems: the physical aspect. For software systems the hardware platform is just a means to an end. Without hardware, software systems are not able to run. However, this aspect is in general not relevant for users of software system interfaces because the underlying hardware usually does not affect the functional behavior of software.

For devices, the physical aspect is essential. In contrast to software systems, the embedded software (but not the hardware) is a means to an end. Software ensures the operability of devices or enables the communication to other systems. However, in general, the user experiences the hardware, not the software.

Thus, the characteristics of devices are split into two views:

- Devices as software systems
- Devices as physical objects

Considering devices as software systems, the first two device characteristics (DC) are:

- *DC1*: Proprietary software interface
- *DC2*: Proprietary data model

This is due to the fact that embedded software on devices is usually not designed to be interoperable [19].

Physical objects can be moved, touched and replaced. Moving an object influences its locality. Touching an object can influence its behavior. In the context of devices such physical influences can trigger specific functions or enable/disable the whole device. In addition, physical objects might not be concurrently usable; thus, they can be reserved. Another aspect is the hardware interface. Software systems as artificial constructs are not equipped with hardware interfaces themselves. The underlying hardware platform typically provides a standardized network interface. Devices are often equipped with serial or proprietary hardware interfaces. Thus, when integrating devices, the hardware interface is a relevant aspect. Another characteristic of devices is the fact that embedded software often cannot be changed or is not even allowed to be changed (e.g., medical devices) [13].

In summary, following additional characteristics of devices can be identified:

- *DC3*: Mobility
- *DC4*: Locality
- *DC5*: Manual influenceability
- *DC6*: Replaceability
- *DC7*: Devices as resource
- *DC8*: Hardware interfaces
- *DC9*: Software changeability

In the next section, these characteristics are analyzed with regard to their impacts on the SODA concept.

## 4. SODA design problems

The nine characteristics of devices identified in the last section cause several SODA design problems on and between the three layers. The Device Layer itself is not affected because the integration of devices starts between the Device Layer and the Device Service Layer. The remaining layers and intermediate layers are explored in the next sections. The identified SODA design problems are expressed in the form of questions. Figure 2 shows where the design problems are located within the system of devices, device service, service registry and service customers.

### 4.1. Intermediate layer 1

Intermediate Layer 1 identifies the layer between the Device Layer and Device Service Layer. Within this intermediate layer, the connection of devices to device services is realized. This includes hardware as well as software interfaces. Taking into consideration the device characteristics DC1, DC8 and DC9, design problem 1 arises.

*Design problem 1: How can the connection of devices to device services be realized?*

### 4.2. Device service layer

Services consist of a service contract and a service implementation [8]. When defining the service contract, the use of proprietary device interface definitions (DC1) and data models (DC2) results in a negative coupling of service consumers to devices. The device interface and its data model will rarely change (DC9), but the replacement of a device (DC6) necessitates the adaptation of the service consumer implementation, which is now faced with another service contract. Thus, the following design problem can be deduced:

*Design problem 2: How can negative types of coupling be avoided when defining contracts for device services?*

The service implementation realizes the service contract. In addition, in the context of device services, the handling of the connection to the device must be implemented. Due to the fact that devices might switch from one device service to another (DC3) or can be replaced (DC6), device services are dynamically faced with different devices and therefore different kinds of interfaces. A standardization of device interfaces is in general not possible (DC9). This aspect is summarized in design problem 3:

*Design problem 3: How can device services dynamically handle different kinds of device interfaces?*

Device services differ from software services. There might be a need to logical separate them from software services within the service inventory. Thus, the group of device services could be assigned to a dedicated custodian. This aspect results in design problem 4:

*Design problem 4: How can device services be separated from software services within the service inventory?*

The management of resources is not the purpose of SODA. The integration of functionality into device services for the purpose of booking devices is not

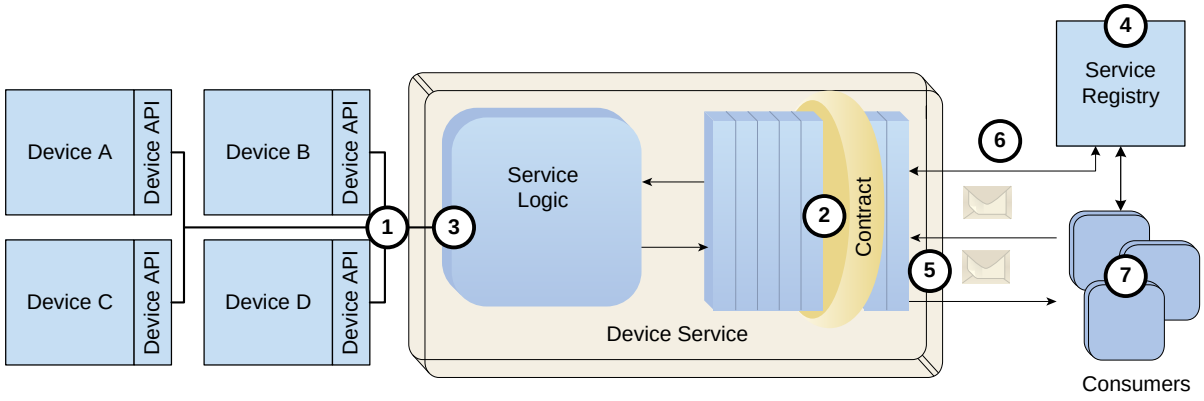


**Figure 2: Location of SODA design problems**

reasonable, due to the fact that the needed device service might be offline. On the one hand, SODA can support a system that manages resources. For example, an automatically generated list of all devices currently available and not in use could be generated. Another example would be the automated collection of device status data for the purpose of maintenance. These aspects do not result in design problems. On the other hand, if a device only supports exclusive usage, but more than one service customer requests access, this conflict must be handled by the device service. Thus, design problem 5 arises:

*Design problem 5: How can device services support the exclusive usage of devices?*

The locality of a device cannot be provided by SODA itself. The locality of a device is either entered manually as a parameter of the device and accessible over the device interface, or it must be dynamically detected by a tracking and tracing system. In both cases, the SODA concept is not directly involved. SODA can enable a standardized access to positioning data, for which purpose, the service contract could contain appropriate interface definitions, e.g., a *getPosition* function. The service implementation decides whether the positioning data can be directly accessed from the device or if other systems need to be called. As a consequence, the service consumer does not need to be concerned about the underlying tracking and tracing mechanism. However, the device characteristic *Locality* (DC4) does not result in a SODA design problem.

### 4.3. Intermediate layer 2

Intermediate Layer 2 identifies the layer between Device Service Layer and Composition Layer. Before services can be combined to a composition, the individual services have to be found. For this purpose, services are published to a service registry [8]. Once registered, services can be found by service consumers, which get all necessary information from the service registry for using the service. In general, services are published manually once. Afterwards, the corresponding registry entry remains unchanged until the service is modified or shut down. Both possible changes are usually well scheduled because service consumer implementations are affected and may not be executable any more. On the contrary, device services can shut down at any time because their operability directly depends on the device. Thus, if a device is out of network coverage (DC3) or switched off (DC5, DC6), the corresponding service immediately will not work any longer. This results in design problem 6:

*Design problem 6: How can the spontaneous appearance and disappearance of device services dynamically be published?*

### 4.4. Composition layer

Service compositions use a set of services to realize specific functionality. If one of the used services does not work, the whole composition is not executable any longer. As mentioned in the last section, the appearance and disappearance of device services can occur anytime (DC3, DC5, DC6); thus, compositions using device services must keep track of the accessibility of the respective services. This aspect is addressed by design problem 7:

*Design problem 7: How can service consumers handle the spontaneous appearance and disappearance of device services?*

## 5. SOA pattern analysis

Several studies were analyzed with regard to SOA design patterns ([3, 4, 9-12, 15]). It has been shown that the SOA design pattern catalog published by Erl [9], in conjunction with several candidate patterns published on his website [10], is the most complete collection of patterns currently available. All SOA-relevant patterns found in other works, were also found in his collection. As a result, 108 SOA design patterns were analyzed in two ways:

a) Which patterns should be used in any case for ensuring a consistent and well designed architecture when realizing the SODA concept? (section 5.1 – 5.4)
b) Which of the patterns identified in point a) are able to resolve SODA-specific design problems? (section 5.5)

The application of SODA results in a set of device services which can be used by other services within a SOA. Thus, every SOA design patterns is potentially applicable, depending on the specific requirements. However, a subset of SOA design patterns can be identified, which should be considered in any SODA project to ensure a consistent and well designed architecture. The intended uses of these patterns are described in the next four sections. Pattern names are put in parentheses and italicized. Next, section 5.5 discusses the SODA design problems. Due to space restrictions, only the results of the analysis are presented and not the complete line of arguments.

### 5.1. Service inventory design patterns

As a first step towards the realization of SODA, device services have to be identified and grouped into a service inventory. Within a service inventory, services

are standardized and governed [9]. A service inventory can range over the whole enterprise (*Enterprise Inventory*) or be limited to a specific subdomain (*Domain Inventory*). In the latter case, several inventories can exist within an enterprise, which can be independently managed.

To avoid redundant service logic, services should be normalized (*Service Normalization*). In the context of SODA, devices with similar functionality can be consolidated to a single device service definition. To separate device services from software services, the inventory can be structured into service layers (*Service Layers*). Three common layers are: the task layer, the entity layer and the utility layer. Device services are dedicated to the utility layer. They enable the access to devices and therefore offer common utility functions (*Utility Abstraction*). The realization of device service logic within different entity services or task services results in redundant implementation. To separate device services from software services, the utility layer can be further partitioned. When using domain inventories, there might be a need for an enterprise wide management of utility services to avoid redundant service implementations. This can be realized by an enterprise wide service layer (*Cross-Domain Utility Layer*).

The standardization of services is useful to avoid unnecessary transformations. For this purpose, uniform data models (*Canonical Schema*) and communication protocols (*Canonical Protocol*) should be introduced. The standardization of data models can additionally be supported by the use of shared schema definitions (*Schema Centralization*). To enable a common understanding of service capabilities and service versions, conventions for service contracts (*Canonical Expression*) and version information (*Canonical Versioning*) should be introduced.

Service contracts have to be published to a service registry (*Metadata Centralization*). In this way, services can be found by both service consumers and service developers who are interested in existing services to avoid redundant implementation of service logic.

## 5.2. Service design patterns

The basic idea of SODA is the encapsulation of devices as services (*Service Encapsulation*). When defining capabilities for a device service, reusability should be taken into consideration to effectively design consumable and composable services. For this purpose, capabilities of a device service should not be designed for a specific problem but for common concerns (*Agnostic Capability*).

By decoupling the service contract from the service implementation, the service implementation can be debugged or optimized in future, without affecting the service consumer (*Decoupled Contract*). If changes to a service contract are necessary, version information should be included into the contract to make consumers aware of possible incompatibility issues (*Version Identification*). In addition, the allowed access to service logic should be limited to the contract (*Contract Centralization*). If subsystems of a service (like devices) can be directly accessed by service consumers, the usage of the corresponding service contract as entry point can be enforced by giving the service exclusive access rights to its subsystems (*Trusted Subsystem*).

The decoupling of device service consumers to devices can be further supported by extracting device specific information (data model, function names, error codes, etc.) from the service contract (*Legacy Wrapper*). In addition to error codes, internal exceptions of the device service implementation should not be transported to the service consumer and replaced by standardized exceptions (*Exception Shielding*).

## 5.3. Service composition design patterns

Being utility services, devices services are designed to be used by other services (*Capability Composition*). For this purpose, capabilities should be designed in a way that enables a maximum of composability (*Capability Recomposition*). In order to realize a loose coupling, services communicate via messages instead of persistent connections (*Service Messaging*). For different purposes (e.g., state data), apart from the message body, additional meta information are placed into the message header (*Messaging Metadata*). In addition, the realization of reliable messaging is usually required (*Reliable Messaging*).

Devices may perform long running activities before sending data back to the device service. Thus, service consumers could be blocked while waiting for the reply message. For this purpose, asynchronous messaging can be used (*Service Callback*), supported by messages queues (*Asynchronous Queuing*). If a service consumer is interested in specific (device) event (e.g., battery low), a subscribing mechanism can be implemented (*Event-Driven Messaging*). If the event occurs, the service informs the service consumer.

## 5.4. Pattern candidates

The usage of a common data format for all message contents is reasonable (*Canonical Data Format*). A specific format is often introduced together with a specific technology (as XML, the Extensible Markup

Language, for web services [25]). However, devices often produce data which have other formats (e.g., pictures) that cannot be transformed. Thus, the support of additional formats (apart from the canonical data format) is necessary (*Alternative Format*). For example, in the case of web services, this can be realized by WS-Attachments [6].

### 5.5. Reflection of SODA design problems

The results of the pattern analysis are summarized in Table 1. Compound patterns are not listed in the table because all individual patterns of compound patterns are included in the table. In the following, we reflect on the SODA design problems.

*Design problem 1*. No pattern could be identified that contributes to this problem. This is due to the fact that existing patterns do not consider devices.

*Design problem 2*. This problem can be resolved by the application of the *Legacy Wrapper* pattern. This pattern advocates the wrapping of proprietary functions and data models by a standardized service contract. It is supported by several other patterns, e.g., *Canonical Schema*.

*Design problem 3*. Services that access underlying resources with dynamically changing interfaces are highly unusual. As a consequence, no pattern could be identified that contributes to this problem.

*Design problem 4*. The separation of device services and software services can be realized by the use of the *Service Layers* pattern and the *Utility Abstraction* pattern. As a result, all device services are assigned to the utility layer. The further portioning of this layer separates device services from software utility services (cf 5.1).

*Design problem 5*. For exclusive usage of devices, the direct access of service consumers to devices must be avoided. This can be realized by the *Contract Centralization* pattern and the *Trusted Subsystem* pattern. Subsequently, two scenarios can be distinguished: a) parallel access to the device is desirable, but not supported by the device, b) parallel access is undesirable or not reasonable for the specific device. In the former scenario, the device service manages incoming requests and realizes a quasi-parallel access to the device (analogous to CPU-Scheduling [24]). In the latter scenario, the device service denies requests if the device is occupied. Alternatively, requests are accepted and processed sequentially (analogous to batch processing [24]). To avoid the blocking of service consumers, the *Service Callback* pattern and *Asynchronous Queuing* pattern can be applied. If the service consumer is only interested in specific events, continuous polling can be avoided by the *Event-Driven Messaging* pattern.

*Design problem 6*. The spontaneous appearance and disappearance of services is highly unusual. As a consequence, no pattern could be identified that contributes to this problem.

*Design problem 7*. For the same reason as with design problem 6, as well as for design problem 7, no pattern could be identified.

**Table 1: SOA pattern analysis with respect to SODA**

| **Pattern Name and Contributor(s)** | **Obligatory for SODA** | **Addressed Design Problem Nr.** |
|---|---|---|
| **Service Inventory Design Patterns** | | |
| Enterprise Inventory (Erl) | ☒ | |
| Domain Inventory (Erl) | ☒ | |
| Service Normalization (Erl) | ☒ | |
| Logic Centralization (Erl) | | |
| Service Layers (Erl) | ☒ | 4 |
| Canonical Protocol (Erl) | ☒ | |
| Canonical Schema (Erl) | ☒ | |
| Utility Abstraction (Erl) | ☒ | 4 |
| Entity Abstraction (Erl) | | |
| Process Abstraction (Erl) | | |
| Process Centralization (Erl) | | |
| Schema Centralization (Erl) | ☒ | |
| Policy Centralization (Erl) | | |
| Rules Centralization (Erl) | | |
| Dual Protocols (Erl) | | |
| Canonical Resources (Erl) | | |

| | | |
|---|---|---|
| State Repository (Erl) | | |
| Stateful Services (Erl) | | |
| Service Grid (Chappell) | | |
| Inventory Endpoint (Erl) | | |
| Cross-Domain Utility Layer (Erl) | ☒ | |
| Canonical Expression (Erl) | ☒ | |
| Metadata Centralization (Erl) | ☒ | |
| Canonical Versioning (Erl) | ☒ | |
| **Service Design Patterns** | | |
| Functional Decomposition (Erl) | | |
| Service Encapsulation (Erl) | ☒ | |
| Agnostic Context (Erl) | | |
| Non-Agnostic Context (Erl) | | |
| Agnostic Capability (Erl) | ☒ | |
| Service Facade (Erl) | | |
| Redundant Implementation (Erl) | | |
| Service Data Replication (Erl) | | |
| Partial State Deferral (Erl) | | |
| Partial Validation (Orchard, Riley) | | |
| UI Mediator (Utschig, Maier, Trops, Normann, Winterberg) | | |
| Exception Shielding (Hogg, Smith, Chong, Hollander, Kozaczynski, Brader, Delgado, Taylor, Wall, Slater, Imran, Cibraro, Cunningham) | ☒ | |
| Message Screening (Hogg, Smith, Chong, Hollander, Kozaczynski, Brader, Delgado, Taylor, Wall, Slater, Imran, Cibraro, Cunningham) | | |
| Trusted Subsystem (Hogg, Smith, Chong, Hollander, Kozaczynski, Brader, Delgado, Taylor, Wall, Slater, Imran, Cibraro, Cunningham) | ☒ | 5 |
| Service Perimeter Guard (Hogg, Smith, Chong, Hollander, Kozaczynski, Brader, Delgado, Taylor, Wall, Slater, Imran, Cibraro, Cunningham) | | |
| Decoupled Contract (Erl) | ☒ | |
| Contract Centralization (Erl) | ☒ | 5 |
| Contract Denormalization (Erl) | | |
| Concurrent Contracts (Erl) | | |
| Validation Abstraction (Erl) | | |
| Legacy Wrapper (Erl, Roy) | ☒ | 2 |
| Multi-Channel Endpoint (Roy) | | |
| File Gateway (Roy) | | |
| Compatible Change (Orchard, Riley) | | |
| Version Identification (Orchard, Riley) | ☒ | |
| Termination Notification (Orchard, Riley) | | |
| Service Refactoring (Erl) | | |
| Service Decomposition (Erl) | | |
| Proxy Capability (Erl) | | |
| Decomposed Capability (Erl) | | |
| Distributed Capability (Erl) | | |
| **Service Composition Design Patterns** | | |
| Capability Composition (Erl) | ☒ | |
| Capability Recomposition (Erl) | ☒ | |
| Service Messaging (Erl) | ☒ | |
| Messaging Metadata (Erl) | ☒ | |

| | | |
|---|---|---|
| Service Agent (Erl) | | |
| Intermediate Routing (Little, Rischbeck, Simon) | | |
| State Messaging (Karmarkar) | | |
| Service Callback (Karmarkar) | ☒ | 5 |
| Service Instance Routing (Karmarkar) | | |
| Asynchronous Queuing (Little, Rischbeck, Simon) | ☒ | 5 |
| Reliable Messaging (Little, Rischbeck, Simon) | ☒ | |
| Event-Driven Messaging (Little, Rischbeck, Simon) | ☒ | 5 |
| Agnostic Sub-Controller (Erl) | | |
| Composition Autonomy (Erl) | | |
| Atomic Service Transaction (Erl) | | |
| Compensating Service Transaction (Utschig, Maier, Trops, Normann, Winterberg, Loesgen, Little) | | |
| Data Confidentiality (Hogg, Smith, Chong, Hollander, Kozaczynski, Brader, Delgado, Taylor, Wall, Slater, Imran, Cibraro, Cunningham) | | |
| Data Origin Authentication (Hogg, Smith, Chong, Hollander, Kozaczynski, Brader, Delgado, Taylor, Wall, Slater, Imran, Cibraro, Cunningham) | | |
| Direct Authentication (Hogg, Smith, Chong, Hollander, Kozaczynski, Brader, Delgado, Taylor, Wall, Slater, Imran, Cibraro, Cunningham) | | |
| Brokered Authentication (Hogg, Smith, Chong, Hollander, Kozaczynski, Brader, Delgado, Taylor, Wall, Slater, Imran, Cibraro, Cunningham) | | |
| Data Model Transformation (Erl) | | |
| Data Format Transformation (Little, Rischbeck, Simon) | | |
| Protocol Bridging (Little, Rischbeck, Simon) | | |
| **Pattern Candidates** | | |
| Alternative Format (Balasubramanian, Webber, Erl, Booth, Riley) | ☒ | |
| Blind Messaging Routing (Erl) | | |
| Canonical Data Format (Erl) | ☒ | |
| Canonical Policy Vocabulary (Erl) | | |
| Composition Endpoints (Erl) | | |
| Composition Extension (Erl) | | |
| Enterprise Domain Repository (Lind) | | |
| Entity Data Abstraction (Erl) | | |
| Entity Linking (Balasubramanian, Webber, Erl, Booth) | | |
| Federated Identity (Wilhelmsen, Rischbeck) | | |
| Forwards Compatibility (Orchard) | | |
| Idempotent Capability (Wilhelmsen, Pautasso) | | |
| In-Memory Fault-tolerant Collection (Chappell) | | |
| In-Memory Fault-tolerant Stateful Services (Chappell) | | |
| Layered Redirect (Balasubramanian, Webber, Erl, Booth) | | |
| Load Balanced Stateful Services (Chappell) | | |
| Policy Enforcement (Little, Rischbeck, Simon, Erl) | | |
| Relaxed Service Implementation (Wilhelmsen) | | |
| Service Virtualization (Roy) | | |
| Transport Caching (Balasubramanian, Webber, Erl, Booth) | | |
| UI Agnostic Service (Roy) | | |
| Uniform Contract (Balasubramanian, Webber, Erl, Booth) | | |
| Validation by Projection (Orchard) | | |

## 6. New SOA design pattern candidates for SODA

For the purpose of resolving remaining SODA design problems identified in the last section, we propose seven new SOA design patterns. A detailed description of the new patterns is out of the scope of this paper, and is the objective of further publications. In addition, different implementation strategies for realizing the patterns are currently analyzed. Thus, in the following only the general abstracted solution ideas are presented.

*Integrated Adapter*. The first three pattern candidates address design problem 1. They are deduced from several existing studies on SODA. The *Integrated Adapter* pattern integrates adapter logic into the device itself. Thus, the device offers its functionality as service, e.g., as realized with the Devices Profile for Web Services (DPWS) in [2]. However, in most cases the manipulation of embedded software is not possible (DC9). Thus, the next two patterns suggest solutions without affecting the device software.

*External Adapter*. The *External Adapter* plugs a hardware adapter onto the device, which a) exports the device functionality as service, and b) enables the networkability of the device (e.g., if the device is equipped with a serial hardware interface only). This was successfully realized by using a XPORT adapter in [14].

*Server Adapter*. Small hardware adapters are often restricted in their processing power and memory capacity. Thus, the implementation has to be realized in low level programming languages. The *Server Adapter* pattern realizes the device integration by plugging the device into a server (directly or over network). Thus, high level programming languages can be used and several devices can be managed by one server. This way of integrating devices was chosen in [23] by using web service technologies.

*Dynamical Adapter*. This pattern addresses design problem 3. It introduces additional adapter logic, which a) identifies the specific devices and b) selects an appropriate adapter. As a consequence, the service logic doesn't have to be adapted to specific integration scenarios. It communicates with the adapter logic in the same way in all scenarios.

*Auto-Publishing*. This pattern addresses design problem 6. It advocates the introduction of a mechanism into the service logic, which automatically publishes and unpublishes the device service to the service registry. When using a UDDI registry (Universal Description, Discovery and Integration), service consumers can be informed about changes by using the UDDI subscriber mechanism [21].

*Standardized Device Service*. This pattern is a compound pattern, i.e., it is comprised of combinations of design patterns [9]. Two established and two new patterns are included:

- Service Encapsulation
- Legacy Wrapper
- Dynamical Adapter
- Auto-Publishing

This pattern combines all patterns necessary for realizing device services. The three integration adapter patterns create adapters, which are selected and used by the device service or may be part of the adapter logic, but do not affect the design of the device service. The *Device Concentrator* pattern also does not affect the device service design. Thus, these patterns are not included into the *Standardized Device Service* pattern.

*Device Concentrator*. This pattern addresses design problem 7. It advocates the establishment of a service, which monitors the availability of devices that meet specific criteria. For example, in hospitals a device concentrator service could monitor all infusion pumps assigned to a specific patient. This consists of a) recognizing new pumps, b) recognizing pumps that are no longer available, and c) collecting data of all available pumps. Thus, service consumers do not need to implement these functionalities themselves. This avoids redundant service logic by extracting common utility logic, as advocated by the *Utility Abstraction* pattern.

## 7. Conclusion and future research

This paper has presented seven specific SODA design problems. Several existing SOA design patterns were analyzed for their applicability for SODA, and especially for their ability to resolve the identified design problems. Three design problems could be resolved by existing SOA design patterns, and for the remaining design problems, seven new candidate patterns were proposed.

The collection of the identified SOA design patterns suitable for SODA in combination with the new candidate patterns could be seen as a pattern language for SODA [1]. However, only the vocabulary (the patterns), and not the grammar (interrelations between patterns and useful pattern sequences) has been defined up to now. Thus, the new pattern candidates need to be discussed, reviewed, practically tested, evaluated and improved. There is undoubtedly much room for further research concerning SOA design patterns for SODA. For this reason, our research agenda contains following points:

- Practical application of the patterns on a real

scenario
- Exploration of different implementation strategies for realizing the patterns
- Detailed publication of the new candidate patterns for scientific discussion
- Evaluation and improvement of the new candidate patterns
- If necessary, development of additional design patterns

The SODA concept is a promising approach, and the next logical step in the SOA field after software services. Hopefully, research about SODA-specific patterns will enable practical applications of the concept by providing proven solutions for specific design problems.

## 8. References


[1] C. Alexander, The timeless way of building. New York: Oxford University Press, 1979.

[2] H. Bohn, A. Bobek, and F. Golatowski, "SIRENA - Service Infrastructure for Real-time Embedded Networked Devices: A service oriented framework for different domains," in International Conference on Networking, International Conference on Systems and International Conference on Mobile Communications and Learning Technologies (ICN/ICONS/MCL 2006), 2006.

[3] F. Buschmann, Pattern-orientierte Software-Architektur: Ein Pattern-System. Bonn: Addison-Wesley.Longmann Verlag, 1998.

[4] S. Conrad, W. Hasselbring, A. Koschel, and R. Tritsch, Enterprise Application Integration - Grundlagen, Konzepte, Entwurfsmuster, Praxisbeispiele. München: Elsevier GmbH, 2006.

[5] S. de Deugd, R. Carroll, K. E. Kelly, B. Millett, and J. Ricker, "SODA: Service Oriented Device Architecture," Pervasive Computing, IEEE, vol. 5, pp. 94-96, 2006.

[6] T. Erl, Servoce-Oriented Architecture - A Field Guide to Integrating XML and Web Services. New Jersey: Pearson Education, 2004.

[7] T. Erl, SOA Principles of Service Design. Boston: Prentice Hall International, 2007.

[8] T. Erl, Web Service Contract Design and Versioning for SOA. Boston: Prentice Hall International, 2008.

[9] T. Erl, SOA Design Patterns. Boston: Prentice Hall International, 2009.

[10] T. Erl, "SOA Patterns - Candidate Pattern List," 2009.

[11] M. Fowler, Patterns of Enterprise Application Integration. Boston: Pearson Education, 2003.

[12] E. Gamma, Design Patterns: Elements of Reusable Object-Oriented Software. Massachusetts: Addison-Wesley Publishing Company, 1995.

[13] A. Gärtner, Medizinproduktesicherheit - Band 1: Medizinproduktegesetzgebung und Regelwerk. Köln: TÜV Media, 2008.

[14] V. Gilart-Iglesias, F. Maciá-Pérez, F. José Mora-Gimeno, and J. V. Berná-Martínez, "Normalization of Industrial Machinery with Embedded Devices and SOA," in Conference on Emerging Technologies and Factory Automation (ETFA '06), 2006.

[15] G. Hohpe and B. Woolf, Enterprise Integration Patterns - Designing, Building, and Deploying Messaging Solutions. Boston: Pearson Education, Inc., 2004.

[16] F. Jammes, H. Smit, J. L. M. Lastra, and I. M. Delamer, "Orchestration of service-oriented manufacturing processes," in 10th IEEE Conference on Emerging Technologies and Factory Automation (ETFA 2005), 2005, pp. 617-624.

[17] D. Krafzik, K. Banke, and D. Slama, Enterprise SOA - Service-Oriented Architecture Best Practices. Indiana, USA: Pearson Education, 2006.

[18] H. Krcmar, Informationsmanagement: Springer Berlin Heidelberg New York, 2005.

[19] K. Lesh, S. Weininger, J. M. Goldman, B. Wilson, and G. Himes, "Medical Device Interoperability – Assessing the Environment," in Joint Workshop on High Confidence Medical Devices, Software, and Systems and Medical Device Plug-and-Play Interoperability, 2007.

[20] C. Mauro, A. Sunyaev, J. M. Leimeister, and H. Krcmar, "Service-orientierte Integration medizinischer Geräte - eine State of the Art Analyse," in Wirtschaftsinformatik 2009 - Business Services: Konzepte, Technologien und Anwendungen Wien, 2009, pp. 119-128.

[21] OASIS, "UDDI Version 3.0.2." vol. 2009, 2004.

[22] M. Schumacher, Security engineering with patterns: origins, theoretical models, and new applications. Berlin: Springer, 2003.

[23] M. Strähle, M. Ehlbeck, V. Prapavat, K. Kück, F. Franz, and J.-U. Meyer, "Towards a Service-Oriented Architecture for Interconnecting Medical Devices and Applications," in Joint Workshop on High Confidence Medical Devices, Software, and Systems and Medical Device Plug-and-Play Interoperability, 2007.

[24] A. S. Tanenbaum, Moderne Betriebssysteme. München: Pearson Studium, 2009.

[25] W3C, "XML Schema Part 0: Primer Second Edition - W3C Recommendation," 2004.