Nr.: FIN-005-2011

Service Variability Patterns in SOC

Ateeq Khan,Christian Kästner,Veit Köppen,and Gunter Saake

Arbeitsgruppe Datenbanken

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Nr.: FIN-005-2011

Service Variability Patterns in SOC

Ateeq Khan,Christian Kästner,Veit Köppen,and Gunter Saake

Arbeitsgruppe Datenbanken

Technical report (Internet)
Elektronische Zeitschriftenreihe
der Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg
ISSN 1869-5078

Impressum (§ 5 TMG)

Herausgeber:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Der Dekan

Verantwortlich für diese Ausgabe:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Ateeq Khan
Postfach 4120
39016 Magdeburg
E-Mail: ateeq.khan@ovgu.de

http://www.cs.uni-magdeburg.de/Technical_reports.html

Technical report (Internet)
ISSN 1869-5078

Redaktionsschluss: 04.05.2011

Bezug: Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Dekanat

Service Variability Patterns in SOC

Ateeq Khan, Christian Kästner, Veit Köppen, Gunter Saake
ateeq.khan@ovgu.de
christian.kaestner@uni-marburg.de
veit.koeppen@ovgu.de
gunter.saake@ovgu.de

Technical Report

Department of Technical and Business Information Systems,
Faculty of Computer Science,
Otto-von-Guericke University,
Magdeburg, Germany

Service Variability Patterns in SOC

Ateeq Khan [1] , Christian Kästner[2], Veit Köppen[1], Gunter Saake[1]

Abstract

Service-oriented computing (SOC) increases flexibility of IT systems and helps enterprises to meet their changing needs. Different methods address changing requirements in service-oriented environment. Many solutions exist to address variability, however, each solution is tailored to a specific problem, e.g. at one specific layer in SOC. We survey variability mechanisms from literature and summarize solutions, consequences, and possible combinations in a pattern catalogue. Based on the pattern catalogue, we compare different variability patterns and combinations of patterns. Our catalogue helps to choose an appropriate technique for the variability problem at hand and illustrates its consequences in SOC.

[1]University of Magdeburg, Germany
[2]Philipps Universität Marburg, Germany

Contents

List of Figures

List of Tables

1 Introduction

Service-Oriented Computing (SOC) is a paradigm to create information systems and provides flexibility, interoperability, cost effectiveness, and higher quality characteristics [1,2]. The trend of service-usage is increasing in enterprise to support processes [3,4].

Software as a Service (SaaS) is a mechanism to deliver software, hosted as services over the internet to consumers when required [5–7]. Service consumers use available services to perform their business operations, however, they also look to better align their business processes with more flexible services (using customization and preferences). Different factors effect this aim of better alignment between services and business operations. Here, we name few factors, e.g. different functional and non-functional requirements, technology differences (e.g. network in case of slow networks and platform differences), limited processing of the data on specific devices, and storage capacity of results.

However, even in the flexible world of services, *variability* is paramount at all layers. Variability is the ability of a system to extend functionality, modify, customize or configure the system according to requirements [8]. We do not want to provide the same service to all consumers but need to provide customized variants. Consumers want to fine tune services according to their needs and will get a unique behaviour, which is tailored (personalized) for their requirements. Fine-tuning depends on available *features* of the services, where a feature is a domain-abstraction used to describe commonalities and differences [9].

However, variability approaches in SOC are ad-hoc. Many solutions exist; however, each one is tailored and aimed for a specific problem or at a specific layer. Some approaches use simple mechanisms for variability, such as, using if-else structure implementations for variability in services. Others try to prevent bloated results of putting all variability into one service (which also violates the service principle that each service should be an atomic unit to perform a specific task) with various strategies, such as frameworks ([10–12]) and languages-based approaches ([13–15]). A single and perfect-for-all solution does not exist in variability. Such a solution is also unrealistic, due to very different requirements and technologies at different layers. Still, we believe that there are common patterns, and developers do not need to rule out inefficient solutions and reinvent better solutions again and again.

We contribute a catalogue of common variability pattern, designed to help developers to choose a technique for specific variability needs. We survey the literature and abstract from reoccurring problems and individual implementation strategies and layers. We summarize our results in six common patterns for variability in the SOC domain (in general many patterns are even transferable to other domains). The patterns describe the problem and the solution strategy including its trade-offs and general enough to allow different implementation strategies at different SOC layers, but also concrete enough to guide a specific implementation. To help developers decide for the appropriate solution to a variability problem at hand, we discuss trade-offs, limitations and possible combinations of different patterns. To aid understanding, we discuss example scenarios of each pattern with their consequences.

The structure of the report is as follows. Section 2 presents variability patterns in SOC. In Section 3, we evaluate and discuss the combination of patterns. At the end, we provide summary of the paper and presents future work in Section 4.

2 Variability Patterns in SOC

Patterns are used to describe knowledge. Software engineers share their knowledge (and reuse) using design patterns for recurring problems. Patterns are used as guidelines to solve problems. To motivate the context, an example pattern (from [16]) is "Different Chairs" consisting a conflict and resolution. In this pattern, problem is different people prefer different chairs. Suggested solution of the problem is to offer variety of chairs with different properties, e.g. soft, hard, different sizes and materials, so people can choose what they need.

We use the pattern template by Gamma et al. [17] with modification to describe our patterns in SOC domain. Our pattern structure is simple and consists of a pattern name, motivation of the pattern or recurring problem, pattern application, examples, implementation technique or solution for the pattern, and consequences of the pattern.

In our pattern catalogue, we have some general patterns for variability, which can be used in various situations. We discuss some implementation techniques in our patterns (summarized in Table 2). Discussion of all implementation techniques for each pattern is out of scope of this paper. For simplicity, we only give few techniques and examples. In contrast to related pattern catalogues [8, 18, 19], which are focused on product lines, we focus on the SOC domain. We use examples from a sports SaaS application, which are used to manage a sports club. The sports application contains different services to display the matches' results, managing players and members. The sports application can be used for different sports domains by using variability approaches.

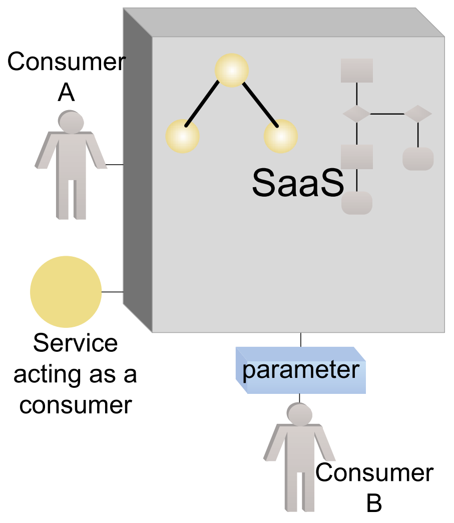

Figure 1: Parameter Pattern

2.1 Parameter Pattern

Motivation: Service providers offer different implementations of a service and selection of services are based on the different values of parameters. Service consumers have different kinds of preferences for a service or need a specific behaviour from services.
Application: This is a simple and widely used pattern. This pattern provides variability solutions based on parameters. Service providers offer variability depending on parameters (depicted in Figure 1) e.g. who is calling the service (consumer id). Access to specific services is decided using this pattern. Service providers plan for variability at design time and this result in variability for a consumer at runtime. Parameters and consumer specific configurations may be bundled together and stored. There are different options to store the configuration of the consumers, mostly stored at the service provider side (although, storage does not have an impact, e.g. at the consumer side or at the service provider side). When a consumer accesses the service, consumer specific configuration is accessed for variability and unique behaviour. For detailed consumer specific modification, configuration files or data are used, to store consumer modifications, e.g. user-interface preferences, domain specific extensions (for bank, health, sports domain, and insurances), or data attributes preferences (e.g. in case of adding or removing database fields). We can also use this pattern for GUI builder for user-interface modification. For workflows, specific workflows, configuration data, or formats can be stored for each consumer separately.
Example: We can use the parameter pattern for sorting, rendering, or for different layouts in a sports service scenario, e.g. offering text commentary of a match based on the consumer language or changing scoring fields for different sports domain. Locale

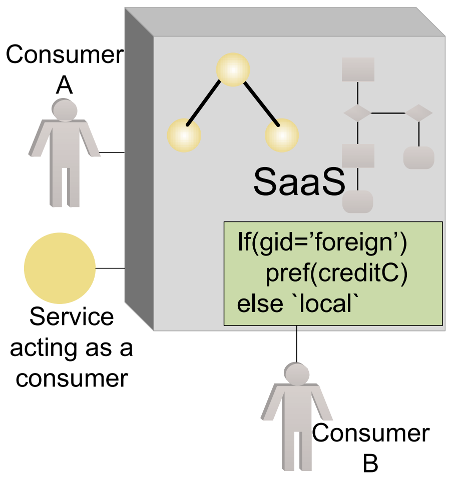

Figure 2: Routing Pattern

settings can also be stored and loaded based on parameters.
Solution: We can store consumer specific settings and parameters as configuration files, e.g. as XML files or stored in a database for each consumer. Parameter storage is also not necessary; a possible extension is passing all required parameters every time when a consumer accesses the SaaS application. There are two types of data associated with this pattern, one is configuration specific data (values configured by consumers for customization), and other is application specific data for each consumer (contain database, values, and users). To present different configuration options, an interface or Integrated Development Environment (IDE) can be used. Instead of presenting all parameters to user, only configuration parameters are presented, and a developer or consumer can configure the attributes from specific behaviour. Configuration data is usually small and less updated as compared to application specific data. For general needs or requirements, configuration data for each consumer can be stored as key-value pair, e.g. consumer id and configuration values (for user-interface favourite colour, selected endpoint, or fields to display).
Consequences: This pattern provides an easy approach to provide variability from the same source code by storing and accessing consumer-specific behaviour based on parameters. Services are selected based on attribute values. Such approach is simple to program and does not require a lot of expertise. This pattern provides flexibility but consumer can choose only from the provided set. Management will be an issue in larger scenarios if parameter conditions are scattered within the code.

2.2 Routing Pattern

Motivation: Even if requirements are same between two consumers, business rules can vary between them. Consumers want to change the business rules such that they follow a specific behaviour. This pattern routes the request based on the rules or consumers requirements.
Application: We can use this pattern for routing requests to different targets, selection of services, changing application behaviour using rules or based on consumer description. Changes can be made at runtime. Flexibility is provided by consumers, providers or by both depending on the scenario and used at different layers. Service providers offer consumers to change the business rules of an application. Rules are used to handle complex scenarios and different conditions. Such conditions are due to user preferences. Meta rules or algorithms can be used to choose which rule has to be executed. Service providers can also allow to use specific operators, e.g. allowing consumers to add if-else branches in the business rules (shown in Figure 2) to control the business logic or using logical operators. Logical operators can also be source of variability, e.g. some consumers may use simple operators and others prefer or require more flexible rules for business logic. We can use this pattern to include/exclude complex service scenarios or to handle exceptions or different scenarios. A considerable amount of literature discusses routing or business rules for adaptation and variability [3, 20]. A consumer can also introduce new business logic in the form of business rules. This pattern is similar to the façade or proxy pattern, discussed in [17] and can be used for the above mentioned functionality.
Example: In our sports system, members pay club membership fees. For payments different options, or routing of services are possible, e.g. local members pay using credit card, bank transfer or both, and foreign members can only pay using credit card.
Solution: Different solutions for implementation do exist for routing. These approaches range from simple if-else statements to complex Aspect-Oriented Programming (AOP) based approaches [20]. Message interception is also be used for routing. We intercept and analyse message to add user-specific behaviour. Different techniques are used to intercept message. Weaving terminology is used for intercepted messages in AOP. Rahman et al. [20] use an AOP-based approach to apply business rules on the intercepted message in SOA domain.

A web service request is intercepted, policy rules are applied on the request, and then it is forwarded to original web service. SOAP header or body are used to carry additional information or enhancements. WS-Addressing [21] uses the SOAP header for additional information. WS-Security [22] uses SOAP header and SOAP body to carry additional security information [4]. Routing can be done by analysing SOAP header or SOAP body and request is routed accordingly. An example scenario is offering different service features from the same service to consumers (e.g. one consumer paying more

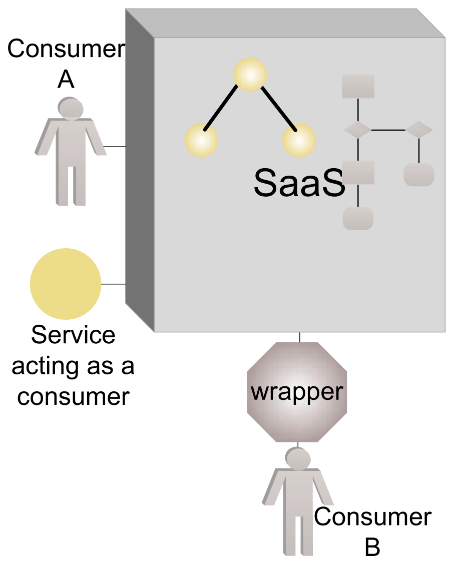

Figure 3: Service Wrapping Pattern

and other paying less or using it for free based on routing rules).
Consequences: Routing allows consumers to use application which suits to their requirements. It also allows to separate business logic from service implementation (for easy modification in rules at runtime). It is also easy to change routing rules and only few changes are necessary. Consumers influence the application behaviour by changing rules.

Adding new business rules or logical operators may add unnecessary loop in an application or inconsistency in application. Validation rules or validators are applied before adding an if-else or branching rule [6,23]. Higher complexity of involved services may lead to inconsistency in application due to rules. Algorithms for validation [24] can also be used to find inconsistent or contradictory rules. Scalability is also an issue for complex applications, routing rules or if-else structures may increase in size and their management become difficult. Routing pattern may introduce single point of failure or decrease in performance in a scenario.

2.3 Service Wrapping Pattern

Motivation: We use this pattern when service is incompatible to use (due to technical or business issue) or provider want to add/hide functionality in services. So, modification is required to use the service in a scenario.
Application: We can use this pattern (as depicted in Figure 3) for the wide variety of

changes, e.g. from technical perspective interface mismatch, message or data transformation, protocols transformation, or for business modifications (content modification). This pattern helps to delegate, modify or extend the functionality for consumers [18,25,26]. Service wrapping can be used to modify existing services and to resolve incompatibilities between services or service interfaces. Services are wrapped and arranged together so that a service delegates the request to other services or component, which implement the service logic. Service wrapping pattern acts between consumers and providers. Service wrapping can be done on the consumer side, provider side, or between consumer and provider. Composite, decorator, wrapper, proxy, and adaptor patterns [17] are similar patterns in object-oriented software reuse with the service wrapping pattern. We can also use this pattern to offer a group of services (from different providers, platforms, or languages) as a composite service to provide sophisticated functionality. Similarly, we can use this pattern to remove/hide functionalities from the services (e.g. in case of composite service, a consumer requires a single service from a workflow instead of a composite service or changes the flow of services in a composite service).
Consumers use the service through the service interface without knowing whether the service provider adds the functionality or hides it from the consumer. We can also use this pattern to support legacy systems without major modification of existing code of the system and exposing functionality as a service [27–29]. The consumers may want to expose her existing systems (containing majority of functionality and business logic) as a service for other consumers, and restrict the access to private business logic from other consumers. Service wrapping can contain the logic for variability as in case of a central rule manager. We can use this pattern to delegate services requests and acting as a central rule manager to other services. Instead of using central rule manager service, we also use this pattern for aspect oriented programming (AOP) based or message interception techniques to delegate the request to other services for variability.

Mostly, this pattern is useful in a mismatch case. Such mismatch cases are due to communication protocols, data types, message formats, interfaces, or interface parameters. An example of such cases are protocols or data formats that do not match between services, e.g. in case of communication protocols where one service uses synchronous communication and the other uses asynchronous communication. In case of asynchronous communication, a consumer request does not wait for the response at the same time. The consumer performs other tasks, and response can be sent back to consumer later or placed in a queue. In asynchronous communication, a queue can be used to store and read the results by the consumer. In synchronous communication, the consumer calls a service and waits for the response.
Example: Combining different services as a composite service may fulfil the consumer demands. A typical example is notifying a consumer about shipping of a parcel by combining shipping service and email notification service. Therefore, for this scenario, two services are wrapped together as a single composite service. An example from

our sports system is offering email and SMS message services (wrapped together as a *notify match* composite service) to send notification about change in match schedule to members and players.

We use service wrapping to add or remove functionalities of services. In case of above-mentioned *notify match* service example, some consumers are only interested in email notification instead of a composite service (with SMS message facility). Providers may offer SMS message service by paying additional fees otherwise hidden by using service wrapper.
Solution: We can use different solutions for this pattern, e.g. using intermediate service, middleware solutions, or tools for variability. To expose legacy systems as service, different techniques are possible, e.g. using service annotations in Java. Intermediate service acts as an interface between incompatible services and contains required implementation logic to overcome the mismatch.

Using SAP Process Integration (SAP PI) [30] as a middleware, different service implementation, workflows, or client interfaces can be used to provide variability. We can use different types of adapters to solve interface mismatch or to connect different systems. When a request from a consumer side is sent to SAP PI, different service implementations, business rules, and interfaces can be selected based on the request. We also use middleware for synchronous-asynchronous communication, in which results are stored at middleware and delivered to the consumers based on their requests. The consumer or provider both (not necessary service owner, could be third party providers) are responsible for variability in this pattern.
Consequences: Using this pattern, we offer different variability solutions. Service wrapping hides the complexity of the scenario from the consumer and simplifies the communication between consumer and composite service (consumers do not care about different interfaces or number of underlying services). Addition or removal of a service becomes easy for the consumer (considered as include/exclude component in case of component engineering). Services are reused and become compatible without changing their implementation details by using service wrapping.

Composite services increase the complexity of the system. Adding services from other providers may effect non-functional properties. Service wrapping increases the number of services (depending on the scenarios composite, adapters or fine-grained) offered from the provider and management of such a system becomes complex.

2.4 Variant/Template Pattern

Motivation: We assume that providers know the consumer variability requirements for specific modules or services. Therefore, providers offer static variants, and consumers

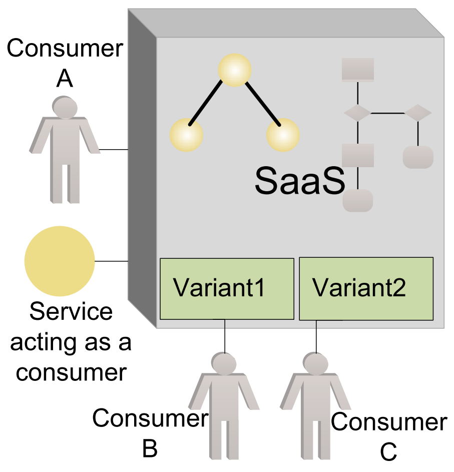

Figure 4: Variant Pattern

configure these variants according to their needs, e.g. variants based on the consumer geographical location, cultural aspects, subscription, consumer groups, and devices.
Application: In this pattern, providers offer a set of service variants to the consumer (as illustrated in Figure 4). These variants are typically generated from the same source code (using generators, product line or by using other options at the service provider side). Service providers plan for the variability and provide variants at design time and consumers select these variants, mostly at runtime. Service providers select the features and varying options based on industry best practices as variants with a pre-defined set of configuration options. Consumers can change the options. In [6, 23], authors suggested to offer a set of templates, so consumers can choose a template in a process or workflow. Templates may contain a single activity or the whole workflow. Consumers can add an activity depending on the template rules (e.g. dependencies and constraints). Variant pattern can also be used for user-interfaces development.
Example: A common example on the web is content management systems, e.g. 40-50% websites content is based on templates [31]. In our sports system, different user-interface variants can be used to display *match score* (suppose in Figure 4, text commentary is offered in *Variant1* for consumer B but online video streaming in *Variant2* from provider side for consumer C.
Solution: The variant pattern is a general pattern and used in various scenarios. Different variants are offered and consumers choose and use options to configure it, e.g. for unique look and feel, workflows, or for viewing/hiding data fields in interface. These configurations are stored as a configuration file and loaded when the consumer invokes the service for customized behaviour. For example, at the user interface level, users are offered different interface options and at design time user configure these options. User specific settings are stored as user-interface configuration and loaded uniquely for

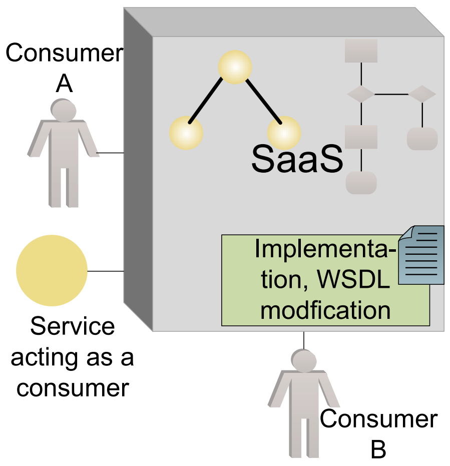
Consumer A
SaaS
Implementa-tion, WSDL modfication
Service acting as a consumer
Consumer B

Figure 5: Extension Points Pattern

each user when she accesses it. We can also combine this pattern with routing pattern to select different variants of services and protocol or message transformation based on some criteria. We can use inheritance, polymorphic approaches, or product line approaches to generate variants of a service at design time [9, 17, 32, 33]. In [9], we discuss how different variants can be offered based on a feature set from the same code base and benefits achieved using variability. WSDL files can also be tailored and used for representing different variants. For user-interface level, Microsoft Silverlight is used. Configuration files are read according to user roles and preferences; afterwards interface is presented to the user. Microsoft Silverlight, XAML, Linq language [34], SaaSDe, and Windows Workflow Foundations are used for interaction with configuration data at interface and data level.
Consequences: This pattern allows to offer optimal solutions in the form of variants. Industry best practices help consumers to choose right options and results in higher quality. The consumer specific options are stored as configuration files.

This pattern does not allow full flexibility to consumers. Developers provide variants in advance and consumers choose only from this set. Managing different variants of a service increases the complexity. Additional information is needed to decide which variant of a service is useful or compatible in a given scenario. Simple variation needs can be handled using variants. Complex scenarios need a flexible platform or architecture, which allows handling of different variants (challenges mentioned in [9]).

2.5 Extension Points Pattern

Motivation: Sometimes, consumers have specific requirements which are not fulfilled by the above mentioned patterns. For instance, consumers want to upload their own implementation of a service, replace part of a service in a process, or upload own business rules to meet the specific requirements. Therefore, service providers offer extension points in a SaaS application.
Application: This pattern requires pre-planning. Service providers prepare the variability as extension points at design time. Consumers provide behaviour at those extension points at runtime. Other consumers access the service without any change. It is similar to the strategy design pattern [17], frameworks, or callbacks (can use inheritance methods at design time). The consumer modifies the application behaviour by uploading implementations, rules, or fine-tuning services (changing service endpoints) according to the requirements. Consumers share the same code base; mostly changes are made at runtime. Extension points allow consumers to add consumer-specific implementations or business logic in the system as shown in Figure 5. A consumer can modify the flow of services in case of workflow or in a composite service. Extension points also enable consumers to replace a part of a composite service or a composite service.
Example: Amazon offers an option to upload code in a separate instance of a service for a specific client in a virtual environment [35]. SAP PI also provides options to upload business rules, Java code, and mapping rules for services. In our sports system, a consumer configures extension point for alternative *scoring* services from different providers using web service endpoint binding method. For *match notifications*, consumers have an option to change the providers of email service or SMS message service activity for notification by changing the service endpoint with desired service endpoint.
Solution: In SOC, service interfaces (WSDL file), service implementations, service bindings, and ports (endpoints) act as extension points in the architecture [36–38]. Consumers can change these extensions points to provide the specific behaviour of an application. For business processes many BPEL engines also provide extension points for transformations using XSLT or XML manipulation techniques [39].

We use physical separation of instances, virtualization, and service binding techniques as solutions for this pattern. In physical separation of instances, providers allocate a dedicated hardware or an instance to the consumer for consumer-specific code execution separately. However, in virtualization, the same hardware server can be used for different consumers requests [40, 41]. Virtualization techniques are used to restrict the scope of the effect of implementations on other consumers. We can also use virtualization to meet non-functional requirements, e.g. in case of high availability, a dedicated virtual instance can be allotted to consumer. Multiple operating system partition with dedicated application and middleware instance runs on shared hardware servers. Each consumer gets her own virtual image of application, middleware, and operating system to execute consumer-specific implementation on a shared hardware [40]. Using virtualization hardware resources are efficiently used and results in low overall hardware costs. Several virtual machines solutions exists (e.g. VMware [3]or XEN [42], OpenVZ [4]) which allows multiple virtual instances of OS on shared hardware. Service provider can configure the OS instance and install the required feature or application. Using virtualization technique consumers can adapt to changes quickly without redeveloping application. Virtualization also offers security and isolation of data for consumers [41,43]. In case of malicious code or failure, only the tenant-specific virtual image will be effected instead of the whole system.

The consumer can also perform modifications for service binding in WSDL. Endpoint modification is a method to modify the service address in a WSDL or in a composite service, e.g. adding an end-point service as an alternative scenario in a web service binding. Endpoint modification can be done at runtime.
Consequences: Extension points offer flexibility to the consumer and allow customization of application behaviour accordingly. There are some potential risks due to offering flexibility through extension points. In a workflow, by allowing a consumer to add activities, it is possible that adding new activities in a workflow introduce loops in application, consuming resources or might result in never ending loops. The consumer's implementation could lead to vulnerabilities or effects other clients. Another problem is allowing a consumer to insert own code, because this may lead to failure of the whole system or instance, e.g. in case of malicious code or virus uploading in the system. Once variability is realised by a consumer, the system must check for the modification (extension points) and test scenarios for correctness of the system, e.g. for resource consumption or effect on the whole process (availability, time constraints for response, etc.)

2.6 Copy and Adapt Pattern

Motivation: Offering variability from the same code base in SaaS is not always a best choice. Consumers have various demands and sometimes, available patterns or approaches fail to fulfil demands from the same code base. Another reason is, if we apply those patterns, management of solutions become complex or result in higher costs as compared to separate service instance.
Application: We use this pattern when service variability techniques do not fulfil consumer demands and shared instance modification for a consumer will harm other consumers. Therefore, a developer copies the service code and modifies it for individual consumer as depicted in Figure 6. This pattern requires source code access for modi-

[3]www.vmware.com
[4]www.openvz.org

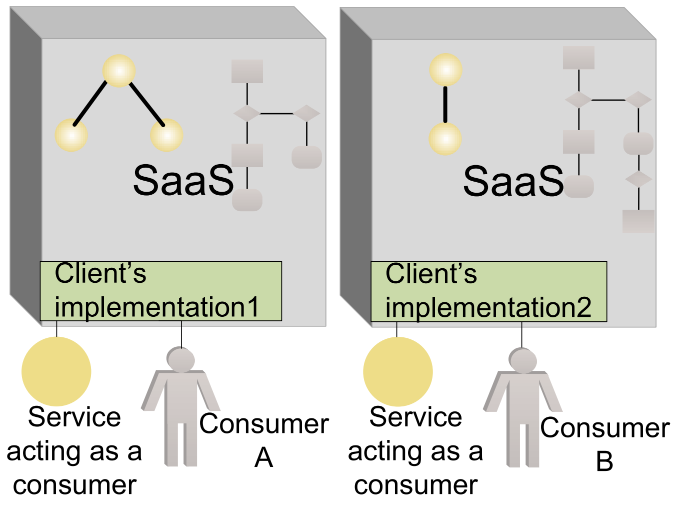

Figure 6: Copy and Adapt Pattern

fication. This pattern allows full flexibility, and consumers can modify or change the service freely. A service instance is deployed later and only specific to a consumer. The consumer decides for the variability and manages itself. Mostly, the consumer is responsible for managing changes or updating the new version of service with own modifications. We also use this pattern where consumers have data privacy issues, e.g. in some countries, data storing, or processing in the shared environment is not feasible.
Example: We use this pattern, if an existing service is specific for a set of consumers, and we need a similar service (with modifications) in other domains (or for other consumers).We use this pattern in scoring service. Scoring is different for football (for consumer group A) as compared to baseball (for consumer group B) and a lot of changes are required, which makes the scenario complex. Open source content management systems (CMS) are examples of such a system in the web domain [29]. Consumers download the CMS copy from the providers and adapt it by modifying the code (or installing other plugins). The consumer does not have to understand the whole code for her modifications.
Solution: Service providers offer a separate instance for a consumer to keep the solution simpler, easier, and to meet consumer demands, although it introduces services with similar codes and functionalities. The consumer can also introduce her own implementation and exposes as a service or modifies the provided solution. In such a case, every consumer gets a customized version of a service instance. This instance is independent from other service instances and other consumers. This pattern provides full flexibility to the consumer for customization and does not effect other consumer processes or instances. We use this pattern at the database or business process layer as well, where a consumer adds or develops her own process in SaaS. In such cases, at the database layer,

a consumer uses a separate database instance to accommodate new database relations and different business requirements.
Consequences: SOC benefits can be achieved in this pattern, although for some parts the application service provider (ASP [44]) model is used, in which each consumer shares the infrastructure facilities but separate server instances. Legacy systems or other applications can be shifted to SOC using this pattern easily. Another benefit of this pattern is that consumers can shift responsibility of the infrastructure and management tasks to the service provider. It is easy to meet changing requirements from different consumers, because it requires modification only in their respective instances. Hence, this pattern offers full flexibility to consumers for customization.

From the provider perspective, this pattern does not scale. It is expensive in terms of costs for the large number of consumers. Number of service or application instances increases very fast. Hardware costs also increase in such cases due to separate instances. Code replication increases the effort for management and decreases productivity.

Furthermore, service maintenance and evolution will be difficult for developers if system is large. Software updates or new version of software must be updated for each instance of tenants manually or individually. Service management (security and monitoring) of such system is also a major cost factor in this pattern. Due to these main problems, it is often not advisable to use this pattern.

3 Patterns Comparison and Combinations

We discuss different patterns and solutions for variability in SOC. In Table 1, we compare these patterns with each other against evaluation factors for variability. Our pattern catalogue covers the common variability problems and solutions in SOC and by no means a comprehensive pattern catalogue.

We identify that some patterns can also be combined together for a better solution or to solve variability problems. For example, *parameter pattern* can be combined with the *extension points pattern* to keep the consumer implementation separate from other consumers. Consumer's implementations are stored in configuration files and retrieved when consumers access the service.

Patterns	Required changes	Flexibility	Scalability	Reusability	Risk	Maintenance	Complexity	Responsibility
Parameters (P)	low	medium	high	low	low	easy	medium	provider
Routing (R)	low	medium	medium	low	medium	easy	medium	both
Service Wrapping (SW)	medium	high	medium	medium	medium	medium	medium	both
Variants (V)	very low	low	medium	high	low	medium	medium	provider
Extension Points (E)	medium	medium	low	low	high	difficult	low	provider
Copy and Adapt (CA)	very high	very high	high	low	low	difficult	low	both
Combining P + E	medium	medium	high	medium	low	low	medium	provider
Combining R + SW	medium	high	high	medium	low	medium	medium	consumer
Combining R + V	low	high	medium	medium	low	medium	medium	both
Combining R + E	medium	low	low	low	medium	difficult	high	both
Combining SW + V	medium	medium	high	low	low	medium	medium	provider
Combining V+ E	medium	high	medium	medium	low	medium	medium	provider

Table 1: Pattern Comparison and Combinations

Methods	Required changes	Management	Introduction time	Code Complexity	Flexibility	Patterns where applicable
Separate WSDLs	medium	medium	runtime	easy	high	P,SW,V
Service endpoint modification	low	easy	runtime	easy	medium	P,R,V,
Ports in same WSDL	low	easy	runtime	easy	medium	P,R,SW,V
Extra methods in WSDLs	medium	medium	design	easy	medium	P,R,SW,V
Extra parameter in method	medium	medium	design	medium	medium	P,R,SW,V
Parameters in soap header	low	medium	runtime	medium	low	P,R,SW,V
User-defined data in SOAP message	high	medium	runtime	high	high	P,R,SW,V
Physical separation	high	difficult	design	medium	high	E
Virtualization	high	difficult	design	medium	medium	E,CA

Table 2: Implementation Techniques for Patterns

We can also combine *routing pattern* with *variant pattern* or *service wrapping pattern* to select different variants of services and protocol or message transformation based on some criteria. *Routing pattern* can use with *extension points pattern* to inject routing rules in application (e.g. uploading code containing routing logic). We can also combine *routing pattern* to offer a set of valid rules based on variants. *Service wrapping pattern* can be mixed with *variant pattern* or *routing pattern* to offer different variants of services. Theses variants can be shared between consumers and used for different service flows or to overcome a mismatch at middleware. *Variant pattern* with the *extension points pattern* gives opportunity to restrict the extension points options to valid combination instead of giving consumers flexibility to add random activities. Using this combination, consumers can add activities or rules from offered templates. An example of such an activity in our sports system is a *notification activity*, a consumer can send an email for a match notification but other consumers want to add additional SMS message activity for notification. So SMS message activity can be added in the workflow from templates activity.

It is possible that different patterns fit in a particular environment or problem. Choosing a pattern depends on many factors, e.g. patterns advantages and disadvantages, application scenarios, business needs, architectures, and customers business models. Similarly different implementation techniques are also discussed in Table 2 and can be used for different patterns. In some organization and countries, consumers have legal or organizational issues, restrictions for shared access of applications (despite the efforts for data and processes confidentiality in multi-tenant applications), so the consumer prefers other patterns.

4 Summary and Outlook

We contributed six variability patterns for SOC that can guide developers to solve different variability problems in practice. We discuss trade-offs according to several evaluation criteria to help deciding for the right solution strategy for a problem at hand. Our pattern catalogue helps to reuse solutions strategies in a manageable way.

In future work, we plan to extend our pattern catalogue into a framework that contains decision criteria to choose and manage variability in SOC with specific implementation techniques. We will evaluate our pattern catalogue further in practice to compare performance where more than one patterns can be used at the same time.

Acknowledgement

Ateeq Khan is supported by a grant from the federal state of Saxony-Anhalt in Germany. This work is partially supported by the German Ministry of Education and Science (BMBF), within the ViERforES-II project No. 01IM10002B.

References

[1] Nicolai Josuttis. *SOA in Practice: The Art of Distributed System Design*. O'Reilly Media, Inc., 2007.

[2] Mike P. Papazoglou and Willem-Jan van den Heuvel. Service oriented architectures: approaches, technologies and research issues. *VLDB*, 16(3):389–415, 2007.

[3] Bart Orriens and Jian Yang. A rule driven approach for developing adaptive service oriented business collaboration. In *Services Computing, 2006. SCC '06. IEEE International Conference on*, pages 182 –189, September 2006.

[4] Dimitrios Georgakopoulos and Mike P. Papazoglou, editors. *Service-Oriented Computing*. The MIT Press, 2009.

[5] Mark Turner, David Budgen, and Pearl Brereton. Turning software into a service. *IEEE Computer*, 36(10):38–44, 2003.

[6] Gianpaolo Carraro and Frederic T. Chong. Software as a service (SaaS): An enterprise perspective. `http://msdn.microsoft.com/en-us/library/aa905332.aspx` last accessed 24.06.2011, October 2006. Microsoft Corporation.

[7] Luis Miguel Vaquero Gonzalez, Luis Rodero-Merino, Juan Caceres, and Maik A. Lindner. A break in the clouds: towards a cloud definition. *Computer Communication Review*, 39(1):50–55, 2009.

[8] Mikael Svahnberg, Jilles van Gurp, and Jan Bosch. A taxonomy of variability realization techniques. *Software - Practice and Experience*, 35(8):705–754, 2005.

[9] Sven Apel, Christian Kästner, and Christian Lengauer. Research challenges in the tension between features and services. In *Proceedings of the ICSE Workshop on Systems Development in SOA Environments (SDSOA)*, pages 53–58, New York, NY, USA, May 2008. ACM.

[10] Javier Cámara, Carlos Canal, Javier Cubo, and Juan Manuel Murillo. An Aspect-Oriented Adaptation Framework for Dynamic Component Evolution. *Electr. Notes Theor. Comput. Sci*, 189:21–34, 2007.

[11] Chang Jie Guo, Wei Sun, Ying Huang, Zhi Hu Wang, and Bo Gao. A framework for native multi-tenancy application development and management. *The 9th IEEE International Conference on E-Commerce Technology*, pages 551–558, 2007.

[12] Woralak Kongdenfha, Régis Saint-Paul, Boualem Benatallah, and Fabio Casati. An aspect-oriented framework for service adaptation. In *ICSOC*, volume 4294 of *Lecture Notes in Computer Science*, pages 15–26. Springer, 2006.

[13] Benjamin Blau, Steffen Lamparter, and Steffen Haak. remash! - Blueprints for RESTful Situational Web Applications. In *Proceedings of the 2nd Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web*, Madrid, Spain, April 2009.

[14] Anis Charfi and Mira Mezini. AO4BPEL: An aspect-oriented extension to BPEL. *World Wide Web*, 10(3):309–344, 2007.

[15] Michael zur Muehlen and Marta Indulska. Modeling languages for business processes and business rules: A representational analysis. *Information Systems*, 35:379–390, 2010.

[16] Christopher Alexander. *A Pattern Language: towns, buildings, construction*. Number 2 in Center for Environmental Structure series. Oxford University Press, New York, 1977.

[17] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Massachusetts, 1995.

[18] N. Yasemin Topaloglu and Rafael Capilla. Modeling the variability of web services from a pattern point of view. In Liang-Jie Zhang, editor, *Proceedings of the European Conference on Web Services ECOWS*, volume 3250 of *Lecture Notes in Computer Science*, pages 128–138. Springer, 2004.

[19] Markus Völter. Handling variability. In *Inproceedings of 14th Annual European Conference on Pattern Languages of Programming (EuroPLoP), Workshop proceedings*, volume 566, pages E5–1–E5–12, 2009.

[20] Syed Saif ur Rahman, Ateeq Khan, and Gunter Saake. Rulespect: Language-Independent Rule-Based AOP Model for Adaptable Context-Sensitive Web Services. In *36th Conference on Current Trends in Theory and Practice of Computer*

Science (Student Research Forum), volume II, pages 87–99. Institute of Computer Science AS CR, Prague, January 2010.

[21] Don Box, Erik Christensen, Francisco Curbera, et al. Web Services Addressing (WS-Addressing). Technical report, W3C, August 2004.

[22] WS-Security Specification. URL: www.oasis-open.org/specs/index.php#wssv1.0, March 2004.

[23] Frederic T. Chong and Gianpaolo Carraro. Architecture strategies for catching the long tail. `http://msdn.microsoft.com/en-us/library/aa479069.aspx` last accessed 24.06.2011, April 2006. Microsoft Corporation.

[24] Domenico Bianculli and Carlo Ghezzi. Towards a methodology for lifelong validation of service compositions. In *Proceedings of the 2nd international workshop on Systems development in SOA environments*, SDSOA, pages 7–12, New York, NY, USA, 2008. ACM.

[25] Holger Mügge, Tobias Rho, Daniel Speicher, Pascal Bihler, and Armin B. Cremers. Programming for Context-based Adaptability: Lessons learned about OOP, SOA, and AOP. In *KiVS 2007 - Kommunikation in Verteilten Systemen - 15. ITG/GI-Fachtagung*, 2007.

[26] Ned Chapin. Research on maintenance characteristics of SOA systems. In *Proceedings of the 3rd International Workshop on a Research Agenda for Maintenance and Evolution of Service-Oriented Systems (MESOA)*. Software Engineering Institute, Carnegie Mellon University, September 2009.

[27] Mike P. Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-oriented computing. *Communications of the ACM*, 46:25–28, 2003.

[28] Qi Yu, Xumin Liu, Athman Bouguettaya, and Brahim Medjahed. Deploying and managing web services: issues, solutions, and directions. *The VLDB Journal*, 17(3):537–572, 2006.

[29] Hanafi Mughrabi. Applying SOA to an ecommerce system. `http://www2.imm.dtu.dk/pubdb/p.php?5496` last accessed 05.05.2011, 2007. Master thesis.

[30] SAP NetWeaver Process Integration. `http://www.sdn.sap.com/irj/sdn/nw-pi71` last accessed 15.03.2011. SAP.

[31] Pankaj Gulhane, Rajeev Rastogi, Srinivasan H. Sengamedu, and Ashwin Tengli. Exploiting content redundancy for web information extraction. *Proc. VLDB Endow.*, 3:578–587, September 2010.

[32] Mike P. Papazoglou and Benedikt Kratz. Web services technology in support of business transactions. *Service Oriented Computing and Applications*, 1(1):51–63, March 2007.

[33] Christoph Pohl, Andreas Rummler, et al. Survey of existing implementation techniques with respect to their support for the requirements identified in m3. 2, July 2007. AMPLE (Aspect-Oriented, Model-Driven, Product Line Engineering), Specific Targeted Research Project: IST- 33710.

[34] Don Box and Anders Hejlsberg. LinQ: .NET language-integrated query. `http://msdn.microsoft.com/en-us/library/bb308959.aspx` Last accessed 04.04.2011.

[35] Amazon Elastic Computing Cloud (EC2). `http://aws.amazon.com/ec2/` last accessed 03.02.2011. Amazon Inc.

[36] Juanjuan Jiang, Anna Ruokonen, and Tarja Systa. Pattern-based variability management in web service development. In *ECOWS '05: Proceedings of the Third European Conference on Web Services*, page 83, Washington, DC, USA, 2005. IEEE Computer Society.

[37] Oliver Moser, Florian Rosenberg, and Schahram Dustdar. Non-intrusive monitoring and service adaptation for WS-BPEL. In *WWW*, pages 815–824. ACM, 2008.

[38] Abdelkarim Erradi, Piyush Maheshwari, and Vladimir Tosic. Policy-driven middleware for self-adaptation of web services compositions. In Maarten van Steen and Michi Henning, editors, *Middleware 2006*, volume 4290 of *Lecture Notes in Computer Science*, pages 62–80. Springer, 2006.

[39] Bill Eidson, Jonathan Maron, Greg Pavlik, and Rajesh Raheja. SOA and the future of application development. In *Proceedings of the First International Workshop on Design of Service-Oriented Applications (WDSOA05)*, pages 1–8. IBM Research Division, IBM, November 2005.

[40] Changhua Sun, Le He, Qingbo Wang, and Ruth Willenborg. Simplifying service deployment with virtual appliances. In *Proceedings IEEE International Confernce Services Computing SCC '08*, volume 2, pages 265–272, July 2008.

[41] Hai Jin, Shadi Ibrahim, Tim Bell, Li Qi, Haijun Cao, Song Wu, and Xuanhua Shi. *Tools and Technologies for Building Clouds*, chapter 1, pages 3–20. Computer Communications and Networks. Springer, 2010.

[42] David E. Williams and Juan R. Garcia. *Virtualization with Xen: including XenEnterprise, XenServer, and XenExpress*. Syngress Publishing, 2007.

[43] Paul Ruth, Xuxian Jiang, Dongyan Xu, and Sebastien Goasguen. Virtual distributed environments in a shared infrastructure. *Computer*, 38:63–69, May 2005.

[44] Mary Celia Lacity and Rudy A. Hirschheim. *Information Systems Outsourcing; Myths, Metaphors, and Realities*. John Wiley & Sons, Inc., 1993.