# Runtime Variability in Online Software Products: A Comparison of Four Patterns

Jaap Kabbedijk, Slinger Jansen, and Thomas Salfischberger
Department of Information and Computing Sciences
Utrecht University, The Netherlands
Princetonplein 5, 3584 CC, Utrecht
J.Kabbedijk@uu.nl, Slinger.Jansen@uu.nl, Tomas@salfischberger.nl

***Abstract*—Business software is increasingly moving towards the cloud. Because of this, variability of software in order to fit requirements of specific customers becomes more complex. This can no longer be done by directly modifying the application for each client, because of the fact that a single application serves multiple customers in the Software-as-a-Service paradigm. A new set of software patterns and approaches are required to design software that supports runtime variability. This paper presents two patterns to solve the problem of dynamically adapting functionality of an online software product; the Component Interceptor Pattern and the Event Distribution Pattern. Additionally, it presents two patterns to dynamically extent the data model; the Datasource Router Pattern and the Custom Property Object Pattern. The patterns originate from case studies of current software systems and are reviewed by domain experts. An evaluation of the patterns is performed in terms of security, performance, scalability, maintainability and implementation effort, leading to the conclusion that the Component Interceptor Pattern and Custom Property Object Pattern are best suited for small projects, making the Event Distribution Pattern and Datasource Router Pattern best for large projects.**

***Keywords*—*architectural patterns, quality attributes, software architecture, variability.***

## I. Introduction

This research has previously been published as conference paper [1] and is extended with a pattern description method and presentation and comparison of two dynamic datamodel extension patterns.

Software as a Service (SaaS) is a rapidly growing deployment model with a clear set of advantages to software vendors and their customers. SaaS allows vendors to deploy changes to applications more rapidly, which increases product innovations while reducing support-costs as only a single version is to be supported concurrently [2]. In the SaaS deployment model, a single application serves a large number of customers. These customers are called tenants, which can be a single user or an organisation with hundreds of users. Because all tenants use the same application, the cost of development and setup of the application can be amortized over all contracts.

The multi-tenant deployment model requires the application to be aware of different tenants and their users, for example in separating the data visible to different groups of users. We define multi-tenancy as: “the property of a system where multiple varying customers and their end-users share the system’s services, applications, databases, or hardware resources, with the aim of lowering costs”. Database designs for multi-tenant aware software require specialized architecture principles to accommodate multiple tenants [3]. One of the challenges in multi-tenant application architectures is the implementation of tenant-specific requirements [4]. Variability of software to fit requirements of specific customers can no longer be done by directly modifying the application for each client or product group, as is customary in Software Product Lines [5]. Because a single application serves multiple customers, only one instance of a product exists, making SPL approaches unusable.

Runtime variability in online software products needs to be enables by a degree of configurability. A new set of software patterns and approaches are required to design software that supports runtime variability. The patterns vary in impact on the technical properties of the software like performance and maintainability, impact on the cost-drivers of the SaaS business model, and the requirements they can fulfil. New patterns are needed for both the data level and instance level of the application. We propose two dynamic functionality adaptation patterns to implement variability at instance level and two dynamic datamodel extension patterns to enable variability at data level. All patterns are evaluated and compared in terms of situational suitability.

The concepts of variability and quality attributes are explained in Section II, after which the expert evaluation used is explained in Section III. Section IV explains how patterns are described in this paper, i.e., functional, system and implementation level. The COMPONENT INTERCEPTOR PATTERN and the EVENT DISTRIBUTION PATTERN, two patterns both solving the problem of dynamically adapting functionality of online business software, are presented in Section V. Section VI presents the DATASOURCE ROUTER PATTERN and CUSTOM PROPERTY OBJECT PATTERN, which introduce variability in the datamodel of online software products. All patterns are compared in terms of security, performance, scalability, maintainability and implementation effort. A concluding overview, presenting the best suitability for all patterns van be found in Section VII.

Please note; in the text we set pattern names in SMALL CAPS according to the convention by Alexander et al. [6].

## II. Related Work

**Software Patterns -** Object oriented design patterns were first introduced by Gamma, Helm, Johnson and Vlissides [7] who define design patterns as recurring patterns of classes

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

and communicating objects in many object-oriented systems. They state "each design pattern systematically names, explains, and evaluates an important and recurring design in object-oriented systems". We distinguish the patterns described in this research from the original object oriented design patterns by using the name software design patterns. We intend to describe software design patterns for variability techniques in a multi-tenant context in a similar manner to the object oriented design patterns described by Gamma et al. [7].

Others have, based on the first set of design patterns, researched the best methods for describing and communicating design patterns for later reuse. Evits and Hinchcliffe [8], for example, apply UML to design patterns and proposes a modeling technique based on UML-modeling. The same approach is taken by Mapelsden, Hosking and Grundy [9] in their proposal for the Design Pattern Modeling Language (DPML). DPML provides a method for the specification of design patterns as well as a notation linking the elements of design patterns in DPML to UML model elements. They consider three forms, the pattern specification, the pattern instantiation and the final UML object model of the instantiation. In a later publication, Mapelsden et al. present tool-support for the DPML to automatically transform a pattern specification into a pattern instantiation and to maintain consistency between pattern specification, pattern instantiation and the UML object model [10].

[11] discuss the need of a more formal design pattern description language to support Computer Aided Software Engineering (CASE) tools. They describe previous pattern description languages based on generic UML diagrams annotated with natural language constraints as a problem for CASE tools. However, their main concern is the fact that previous pattern description approaches tend to describe a single implementation of the pattern where the true meaning of the pattern is lost to a description of implementation details. The running example is the Abstract Factory Pattern as described by Gamma et al. [7]. The proposed solution is to apply three separate layers of modeling, the role-model, type-model and class-model. At the highest level of modeling the role-model only describes the parts of a design pattern and their relative roles and interaction. The type-model is a refinement of the role-model where details like implemented methods are added. The type-model should according to Lauder and Kent [11] be supplemented by a textual description of the motivation, trade-offs and known uses. The final refinement of the type-model is the class-model, where a concrete implementation is described as is the case in previous pattern description languages.

**Variability -** The field of software variability has been the subject of research from both the modeling perspective as well as the technical perspective [12]. The application of variability modeling as used in product line variability [13] to software as a service environments has been described by Mietzner, Unger, Titze and Leymann [14]. Variability modeling as discussed in the aforementioned works contributes to the understanding of where the application architecture needs to be able to accommodate change or extension. Patterns play an important role in modeling and solving variability in software products [15].

Svahnberg, van Gurp and Bosch [16] propose feature diagrams as a modeling technique to describe the different variants of feature in a software product. They use their feature diagrams as the basis for a method to identify variability in a product, constrain this variability, pick a method of implementation for the variability and further manage this variability point in the application lifecycle. The main difference from the objectives of our research is that Svahnberg et al. describe implementation techniques for variability per installation instance of the software, whereas we focus on *runtime* variability in a multi-tenant context.

**Quality Attributes -** Benlian and Hess [17] identify *security* as one of the most important risk-factors perceived, followed by performance risks. To assess security risks, SaaS vendors need to include security as a quality attribute in their design of the architecture. This leads to security as the first desired quality attribute for business SaaS. *Performance* as an important factor to SaaS users is closely related to the most important factor, i.e., cost [17]. When performance is insufficient, clients are lost, when the system uses too many resources to gain an acceptable level of performance, cost is increased. A SaaS vendor must thus assess the possible performance impact of changes to the software. To control cost in business SaaS, the SaaS vendor needs to utilize its opportunities for scalability to decrease the cost of hardware or hosting fees (e.g., using scalable software to make optimal use of cloud-hosting).

Another cost driver in SaaS is the *cost of development* and *maintenance* of the software product. Maintenance cost is generally decreased by having to maintain only a single version instead of multiple previous releases. On the other hand, this maintainability cost-saving must not be lost while implementing runtime variability. Thus, scalability and maintainability are also desired quality attributes for business SaaS. Another way the implementation of runtime variability will influence product cost is through implementation-cost. Development is a cost-driver for SaaS, thus if one or more specialized developers are required to implement a certain pattern this will influence the final product cost.

The identified quality attributes are the following:

**Security** - The ability to isolate tenants from each other and the possible impact of security breaches in custom components on other parts of the system.

**Performance** - The utilization of computing, storage and network resources by the application at a certain level of usage by clients.

**Scalability** - The relative increase in capacity achieved by the addition of computing, storage and network resources to the system as well as the flexibility with which these resources could be added to the system.

**Maintainability** - The ease with which the system can be extended and potential problems can be solved.

**Implementation Effort** - The effort required to implement and deploy a specific system.

## III. Research Approach

In order to gather the patterns in this research, a design science approach [18] was used in which the initial solutions are observed in case studies in which one of the authors took part as a consultant. The solutions are implemented in current commercial software products. The architecture description

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

and source code off the software products is examined and checked if runtime variability in functionality or datamodel is supported. Whenever this is the case, the solution used is documented and the consequences of the solution analyzed. Solutions observed in at least three products are presented as patterns and are evaluated by two domain experts to ensure correctness and usefulness. The evaluation of the cases by experts enhances the validity of the cases [19]. During each evaluation session, patterns are discussed with an expert, in a semi-structured way. Standard questions related to the quality attributes are asked, after which issues are freely discussed per quality attribute.

The first expert selected is a senior software architect in an international software consulting firm specialized in large scale development of Enterprise Java applications. His role is to investigate technologies and methodologies to help design better architectures resulting in faster development and more extensible software. A recent project includes a multi-tenant administrative application storing security sensitive data for multiple organizations. The second expert is a technology director and lead architect for an application used in distributed statistics processing of marketing data, previously working in software performance consulting for web-scale systems. His experience lies in the field of high-performance distributed computing. The application his company works on focuses of low-latency coordinated processing of large volumes of data to calculate metrics used for marketing. Performance and scalability are important areas of expertise for their product.

## IV. Pattern Description Method

The use of patterns in order to describe multi-tenant systems is different from the way object oriented design pattern are commonly applied. An object oriented design pattern describes common solutions to problems in object oriented software design. The most important difference between object oriented software design and the design of multi-tenant systems is that the problem scope in multi-tenant systems is not limited to only the objects in object oriented software. The software system is considered not only to be a set of source files, but to include supporting systems like databases, message-bus and infrastructure.

The needs for a description language for the discussed design patterns thus includes the need to describe any necessary characteristics of the supporting systems and auxiliary materials. When considering design patterns for software systems we propose a combination of description techniques at different levels similar to Lauder and Kent [11]. Instead of modelling different levels of detail and abstraction within only object oriented design, different levels of the software architecture including supporting systems have to be modelled. The levels we propose to describe online systems are:

1) Functional level
2) System level
3) Implementation level

**Functional level -** This level describes the functional intention of the pattern in a technical context. Multiple different patterns can share the same model at functional level, because several patterns can be designed to reach the same functional effect with, for example, different performance and scalability characteristics. For the graphical modelling of the functional level, UML class diagrams are used as shown in Figure 1. This diagram captures the functional situation resulting from application of the pattern without considering implementation of pattern instantiation details.

**System level -** This level models the overview of the software including supporting systems after the application of the pattern. Interaction among different components within and between systems as a result of the implemented pattern are

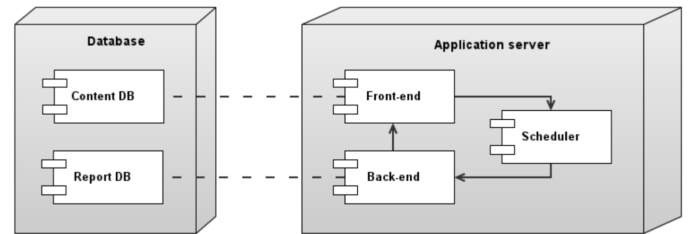


Fig. 2: Example UML Deployment Diagram

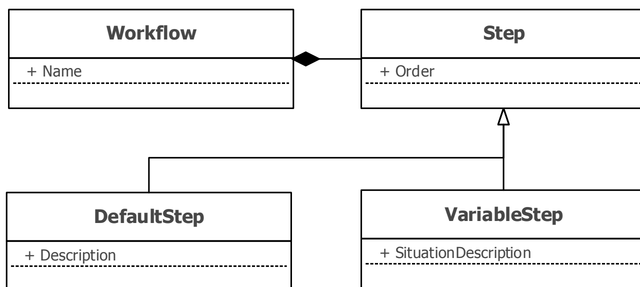


Fig. 1: Example UML class diagram

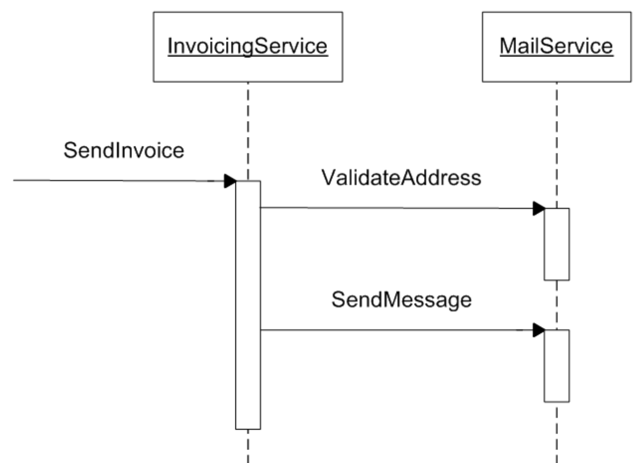


Fig. 3: Example Sequence Diagram

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

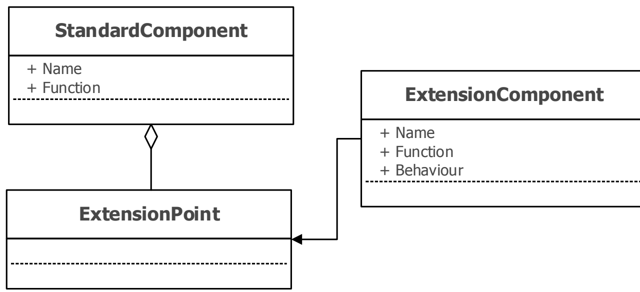


Fig. 4: Functional Model for adapting functionality

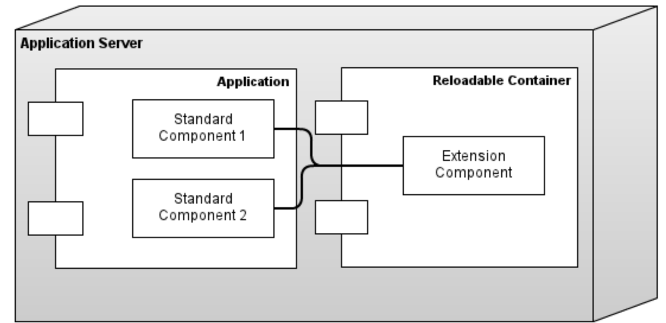


Fig. 5: Component Interceptor Pattern: System Model

shown. A UML deployment diagrams [20] is used to describe this level (see Figure 2 for an example).

**Implementation level -** The third level describes the potential implementation of the pattern. These diagram depicts a specific implementation of the components of the pattern. The implementation diagram is closely related to the system model, but depicts the method of application of the components in the system model on a more detailed level. Within this research we use a sequence diagram as shown in Figure 3 to illustrate the implementation. This description level should be regarded as a possible way to implement the pattern, but it does not prescribe a specific implementation.

## V. Dynamic Functionality Adaptation Patterns

### A. Problem Statement

Software product vendors not only need to offer a *data model* that fits an organisation's requirements, *software functionality* also has to meet an organisation's processes [21]. When tailor-made software is developed, it is possible to set the requirements to exactly match the processes of a specific organisation. For standard online software products this is not possible and differences between requirements of organisation have to be addressed at runtime.

A requirement for the ERP system of a manufacturing company could be to send a notification to the department responsible for transportation if tomorrow's batch will be larger than a certain size. If this requirement is not met by the software product selected, the company could either decide to select another software product or develop a tailor-made application that does meet their requirements.

To allow for the addition of extra functionality in the application, a solution that allows to configure this functionality is needed. This functional situation is modeled in Figure 4, the envisioned functional situation. The *StandardComponent* is a normal component of the software with default functionality, this component has a set of *ExtensionPoints*. An *ExtensionPoint* is a location within the normal workflow where there is a possibility to add or change functionality. This functionality is specified in an *ExtensionComponent*, which contains the actual functionality that is to be executed at the specified *ExtensionPoint*.

Two different patterns are identified, both offering a solution to dynamically adding functionality to a software product.

### B. Component Interceptor Pattern

The Component Interceptor Pattern as depicted in Figure 5 consists of only a single application server. Interceptors are tightly integrated with the application, because they run in-line with normal application code. Before the *StandardComponent* is called the interceptors are allowed to inspect and possibly modify the set of arguments and data passed to the standard component. To do this the interceptor has to be able to access all arguments, modify them or pass them along in the original form. Running interceptors outside of the application requires marshalling of the arguments and data to a format suitable for transport, then unmarshalling by the interceptor component and again marshalling the possibly modified arguments to be passed on to the standard component that was being intercepted. This is impractical and involves a performance penalty [22].

Running the extension components inside the application-server while supporting runtime variability requires support for adding and changing interceptors at runtime. The system model depicts this requirement in the form of a reloadable container. In some implementations this could be as simple as changing a source file, because the programming platform used will interpret source code on the fly. Other platforms require special provisions for reloading code, such as OSGi for the Java platform or Managed Extensibility Framework for the .NET platform.

Figure 6 depicts the interaction with interceptors involved. Interaction with standard components that can be extended goes through the interceptor registry. This registry is needed

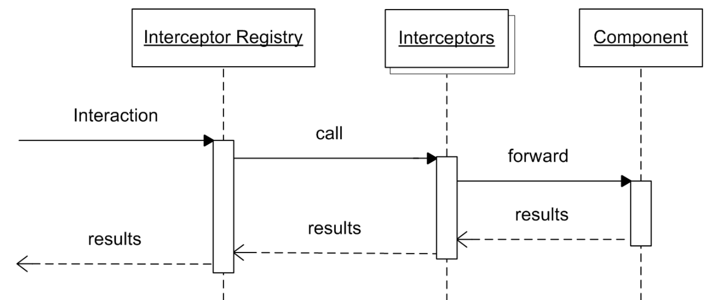


Fig. 6: Component Interceptor Pattern: Sequence Diagram

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

to keep track of all interceptors that are interested in each interaction. Without the registry the calling code would have to be aware of all possible interceptors. As depicted, multiple interceptors can be active per component. It is up to the interceptor registry to determine the order in which interceptors will be called. An example strategy would be to call the first registered interceptor first or to register an explicit order when registering the interceptors.

Each interceptor has the ability to change the data that is passed to the standard component, modify the result returned by the standard component, execute actions before or after passing on the call or even skip the invocation of the next step all together and immediately return. Immediately returning would for example be used when the interceptor implements certain extra validation steps and refuses the request based on the outcome of the validation. As a result of these possibilities the interceptors must be invoked in-line with the standard component, the application cannot continue until all interceptors have finished executing.

### C. Event Distribution Pattern

In the event distribution pattern the application generates events at extension points, which are distributed by a broker. At each extension point the standard component is programmed to send an event indicating the point and appropriate contextual data (e.g., which record is being edited) to a broker. For example in a CRM system the standard component for editing client-records sends a *ClientUpdated* event with the ID of the client that was edited. Extension components listen for these events and take appropriate actions based on the events received. In the example of a *ClientUpdated* event, an extension component could be developed that sends a notification to an external system to update the client details there.

The system model in Figure 7 depicts the distributed nature of the EVENT DISTRIBUTION PATTERN. Standard components run in the application server, sending events to a central broker, which can be run outside of the application. Extension components are isolated and can be on a separate physical server or run as separate processes on the same server depending on capacity and scale of the application. Components are loosely coupled, sharing only the predefined set of events. The standard components are unaware of which extension components listen for their events, execution of extension components is decoupled from the standard components. Executing the extension components separately allows for independent scalability of these components. Depending on system load and the volume of events each component listens for, it is possible to allocate the appropriate amount of resources to each component. Because there is no interaction between listeners, it is possible to execute all listeners in parallel if appropriate for the execution environment.

Standard components publish events to the broker as depicted in the sequence diagram in Figure 8. The activation of the standard component not necessarily overlaps with its listeners. After publishing the event, a standard component is free to continue execution. Depending on the fault tolerance and nature of the events it is up to the standard component to make a trade-off between guaranteed delivery at a higher latency by waiting on the broker system to acknowledge reception of the event or continue without waiting for such an acknowledgement. If, for example, an event is only meant to prime a cache for extra performance the loss of such a message would not impact critical functionality of the system while waiting for the message might mitigate any performance gains. On the other hand, if an event is used for updating an external system for which no other synchronization method is available the system needs guaranteed delivery to function correctly. At design time this decision can be made on an event by event basis depending on the capabilities of the messaging system used.

Because of the one-way nature of events and decoupled execution of extension components it is not possible for an *ExtensionComponent* to stop standard functionality from happening. In the observed system this was solved by allowing *ExtensionComponents* to execute a compensating action in their listener. The compensating action is sent from the listener component back to the system independently of the original action that caused the event. An example of such a compensating action is an extension component that monitors changes to certain records and reverts the change in case special conditions are met. This approach has the added benefit that any changes made by extension components are clearly visible in audit logs, which simplifies tracing possibly unexpected system behaviour back to an *ExtensionComponent*.

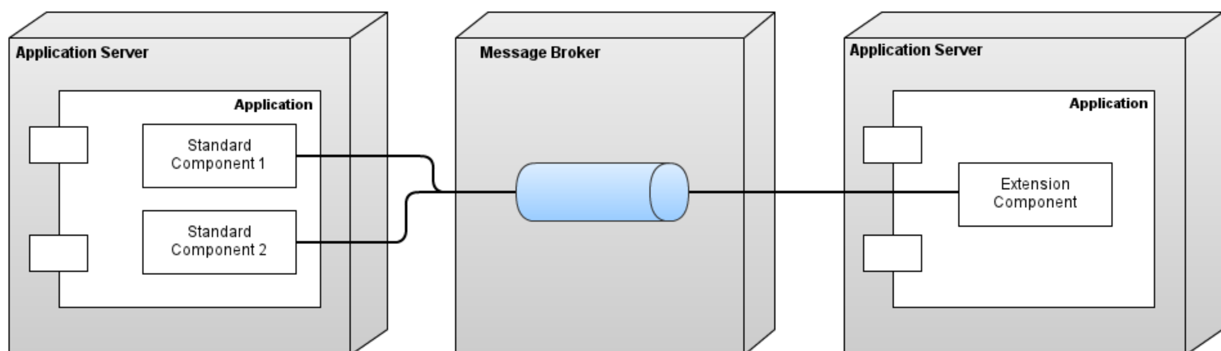


Fig. 7: Event Distribution Pattern: System Model

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

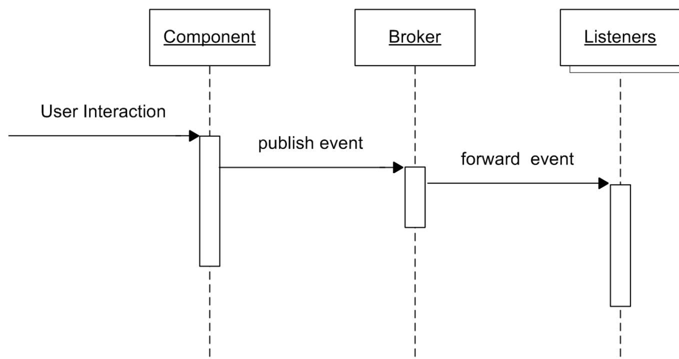


Fig. 8: Event Distribution Pattern: Sequence Diagram

### D. Pattern Comparison

This section presents an analysis of both patterns on the five presented quality attributes.

*1) Security:* When adapting functionality of an application, there is always the possibility of introducing new security vulnerabilities. This is an inherent risk of extending an application. The variability patterns do, however, influence how much larger the attack surface becomes and how well a breach in one of the components is isolated from other components. In the COMPONENT INTERCEPTOR PATTERN, the code handling the new functionality becomes part of the application and will have the ability to execute arbitrary code within the context of the main application as depicted in Figure 5. It will also have full access to any parameters passed to intercepted functions as well as any returned values. A security breach in the extension components (interceptors) is not isolated to only those components, unless extra security measures are implemented to separate the components from the main application. Adding extra security measures, however, does have an impact on the performance efficiency of the application.

The EVENT DISTRIBUTION PATTERN isolates the extension components from the application by executing them in a separate context based on incoming events as depicted in Figure 6. This execution in a separate context allows for more isolation between extension components and the main application components. The components also have far more limited access to standard functionality, because any change the component wants to make has to go through explicitly exported APIs or messages. Combined with event-sourcing, any change to data as a result of custom functionality is fully traceable including the original values [23].

*2) Performance:* The COMPONENT INTERCEPTOR PATTERN executes interceptors within the context of the application. This results in little overhead when executing the extension components, because data does not need to be marshalled, unmarshalled and transferred between applications. However, for security reasons it could be necessary to separate the interceptors from the main application as described in the previous section. This removes one of the performance advantages of the component interceptor pattern because data must be transferred between the different contexts.

Applications implementing the EVENT DISTRIBUTION PATTERN require the setup of a message broker that handles all events coming from the application and going into the extension components. This requires extra processing and network resources and in the case of durable message delivery mechanisms also storage resources reading and writing the messages. To transfer the events from the application via a message broker to the extension components the events must be marshalled into a format suitable for transferring over a network and unmarshalled upon reception by the extension component, these steps add non-trivial cost to the operations.

*3) Scalability:* Applications using the COMPONENT INTERCEPTOR PATTERN will execute interceptors within the context of the application. This has performance advantages described in the previous section, however, the interceptors cannot be scaled independently of the application. When a high number of interceptors exists requiring significant resources the application as a whole needs more application servers to execute. The interceptors must be available to all application servers in that case.

On the other hand, the EVENT DISTRIBUTION PATTERN decouples the execution of the event handlers from the application by running them on a logically separate application server. Because events are handled outside the execution flow of the standard components they can also be distributed to multiple systems. Adding extra application servers subscribing to the same events in the message broker the processing capacity of events could increase linearly. For the EVENT DISTRIBUTION PATTERN this requires a message broker system that is able to handle the increasing numbers of messages. Those systems are available off the shelf from open source projects like Fuse Message Broker, JBoss Messaging, RabbitMQ and commercial offerings like Microsoft BizTalk, Oracle Message Broker or Cloverleaf.

*4) Maintainability:* When adapting the functionality of an application, maintainability is also affected by the necessity to make sure future extensions and modifications are compatible with any custom functionality implemented for tenants. This is a trade-off between the flexibility and depth with which *ExtensionComponents* can affect the application and the impact that changes to the application will have on the *ExtensionComponents*. As an example of the aforementioned trade-off, a simple system with only a single *ExtensionPoint* will have a much lower impact on maintainability than a complex system with a very high number of *ExtensionPoints*. This however affects both patterns equally.

The way the patterns decouple *ExtensionComponents* from *StandardComponents* is however a differentiating factor. In the COMPONENT INTERCEPTOR PATTERN the *ExtensionComponent* is more tightly integrated with the *StandardComponent* because calls to a *StandardComponent* at an *ExtensionPoint* go through the interceptor providing all parameters and return values of the call. When changing calls by adding or removing parameters this will directly affect the input of each *ExtensionComponent* registered from that *ExtensionPoint*. When applying the event distribution pattern the integration is more decoupled because calls to StandardComponents are not directly affected by the *ExtensionComponents*. Instead the *ExtensionComponent* receives a standardized event-message and uses a provided API to send any changes or other actions back to the application. This allows for changes to

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

the *StandardComponent* without changing the event-messages going to the *ExtensionComponent*. At the same time the API used by *ExtensionComponents* to influence the application can be kept stable for small changes or versioned to support future compatibility using methods like the one described by Weinreich, Ziebermayr, and Draheim [24].

*5) Implementation Effort:* When implementing a pattern for adding functionality to an application we distinguish two factors determining the implementation effort. The first factor is the direct effort required to implement the pattern in the system, e.g., adding *ExtensionPoints* to the *StandardComponents* of the application. The second factor is the effort necessary to implement *ExtensionComponents*. Later changes to the components might also require development effort, this is however excluded from implementation effort because it is covered under maintainability. Both patterns require the definition and implementation of *ExtensionPoints*, the way these points are implemented differs per pattern. When implementing the COMPONENT INTERCEPTOR PATTERN it is necessary to setup an Interceptor Registry and modify calls to *StandardComponents* to go through the Interceptor Registry.

In the EVENT DISTRIBUTION PATTERN, a message broker system must be setup to handle the event-messages flowing from *StandardComponents* to *ExtensionComponents*. The application still has to be modified at the ExtensionPoints to send the event-messages belonging to that *ExtensionPoint*. A larger difference between the two patterns emerges in the way they influence the system. Using component interceptor pattern each interceptor has full access to the application because it executes within the same context. Communication with *StandardComponents* from within *ExtensionComponents* could use normal function-calls just like any other part of the system. This differs from the event distribution pattern where the *ExtensionComponents* execute in a separate environment outside the context of the *StandardComponents*. Any interaction between *ExtensionComponents* and *StandardComponents* needs to go through an external interface. Depending on the type of system and the requirements for interaction this requires the development of some sort of (webservice-)API for the *ExtensionComponents* to use.

The second factor of implementation effort, the effort required to implement ExtensionComponents, affects both patterns. In the COMPONENT INTERCEPTOR PATTERN the implementation requires the development of an interceptor, which executes the correct behaviour when certain conditions are met. The EVENT DISTRIBUTION PATTERN requires the development of ExtensionComponents, which listen for the right messages and execute the correct functionality when certain conditions are met.

Please see Table I for an overview of the evaluation of both patterns. Plus and minus signs are used to indicate whether a characteristic is positive or negative. Keep in mind all scores are relative scores compared to the other pattern.

## VI. DYNAMIC DATA MODEL EXTENSION PATTERNS

### A. Problem Statement

Organisations within the same or different market all strive to differentiate themselves, which results in numerous different working processes each with specific requirements for the supporting software systems. Additionally, across markets and jurisdictions differences exist in regulations and standards which require the storage and reporting of different data for each organisation. Organisations will thus set varying requirements to store data specific to their needs. These requirements could be met by software specifically designed for the market in which this organisation operates or even software tailored to the needs of one specific organisation. Specializing software of a small market or even single organisation decreases the number of possible clients for the software vendor and increases the cost per client. A software product that provides enough variability on the data model to meet organisation specific requirements will decrease cost and attract clients that cannot currently be serviced by software products unable to meet their specific requirements. Extension of the data model by creating additional fields to store data that are specific to an organisation or their working processes is a common requirement [25].

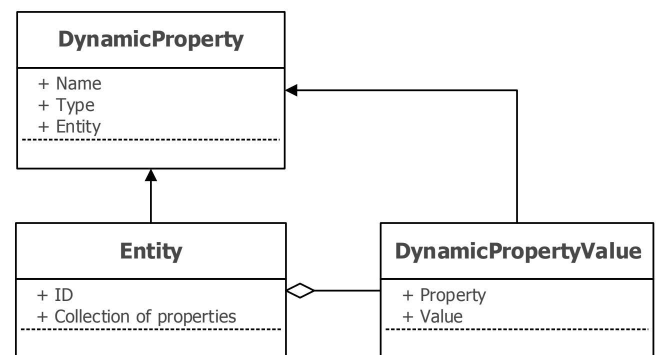


Fig. 9: Functional Model for datamodel extension

In case of standardized software, where this requirement is not met by the default installation of the software, an extension of the existing data model is required. Figure 9 depicts the envisioned functional situation, storing custom properties of entities in the domain model. The depicted *Entity* is the original entity in the application domain model which contains a *DynamicPropertyValue* and has a relation to a *DynamicProperty*. This property is configured for a specific tenant and holds settings like for example a name and expected data-type.

### B. Datasource Router Pattern

In this pattern, the application uses a different database instance (or schema) for each tenant. Custom properties are then added to the database as normal fields. Each component in the application accesses this database through the *Datasource Router*. The *Datasource Router* component determines which database is to be used (based on the tenant the current user belongs to) and routes all access to the right database automatically. The other components can thus work without being aware of the fact that the application is actually serving multiple tenants using different databases.

The system model, which is shown in Figure 10, describes the overview of the system when implementing the DATASOURCE ROUTER PATTERN. As shown, the application uses

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

TABLE I: OVERVIEW OF BOTH DYNAMIC FUNCTIONALITY ADAPTION PATTERNS

| | Component Interceptor Pattern | Event Distribution Pattern |
|---|---|---|
| Security | - Extension components execute within application scope | + Isolation of extension components and full traceability of actions by extension components |
| Performance | + Direct execution of extension components | - Network overhead for calling extension components<br>- The broker system requires extra resources |
| Scalability | - No independent scaling of extension components<br>- Does not scale to high number of extension components | + Independent scaling of extension components<br>+ Extension components cannot delay standard components<br>- Requires scalable message-broker system |
| Maintainability | - Tight coupling of extension components | + Loose coupling of extension components |
| Implementation Effort | + Direct communication with standard components<br>+ Access to all data by design. | - Requires the setup of a message broker system<br>- Requires a separate mechanism to communicate with the application |

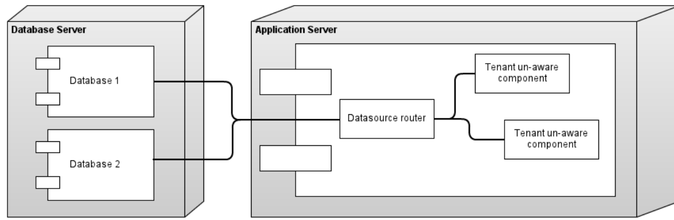


Fig. 10: Datasource Router Pattern: System Model

multiple separate databases (i.e., Database 1 and Database 2 in the figure) to store data for different tenants. Each component accesses the database through a *Datasource Router*, which determines to which database the queries are sent. Due to this isolation the components that access the database never encounter data for multiple tenants at once, since a query will always return results for one and only one tenant, because it is sent to a database that contains only data for a single tenant. This means the components do not need to be multi-tenancy aware in querying the data.

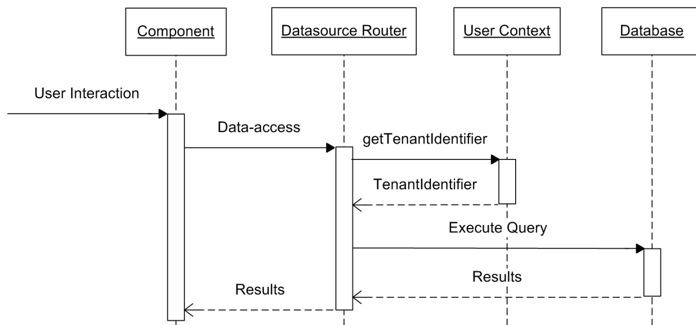


Fig. 11: Datasource Router Pattern: Sequence Diagram

The interaction between tenant-unaware components and the database goes through the *Datasource Router*. The sequence diagram in Figure 11 depicts the interaction from component through *Datasource Router* to the actual database. First, the user interacts with a component, this component requires access to data, which is done through the *Datasource Router*. The *Datasource Router* is then responsible for determining which tenant the current user belongs to, this responsibility is delegated to the *User Context*. It is implementation dependant how this *User Context* is implemented, the only requirement is that it is able to tell the *Datasource Router*, which tenant is to be used in the context of the current request. After determining which tenant is active the *Datasource Router* executes the query on the right database (selected based on the active tenant), the results are then returned to the component, which originally needed access to the data. In this sequence, it is clear that from the perspective of a component requesting data it does not matter how multi-tenancy is implemented in deeper layers. The component is isolated from these choices and the possible complexity involved in selecting the right datasource to use for the current user.

### C. *Custom Property Object Pattern*

When implementing the CUSTOM PROPERTY OBJECT PATTERN, data from all tenants is stored in a single database with a single schema. Any additional data like custom properties is modeled in the design of the application as separate custom property objects, which are stored in the existing static schema. Because all data is stored in a single database components using that data need to be aware of multi-tenancy and explicitly query for data of a specific tenant.

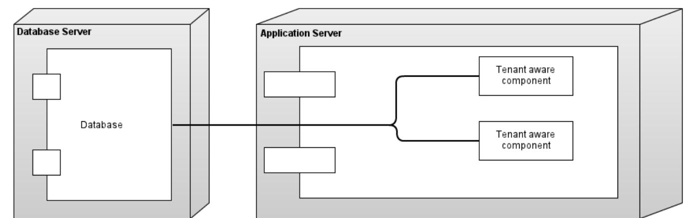


Fig. 12: Custom Property Object Pattern: System Model

This pattern prescribes the storage of all data in a single database, which is accessed by components that are aware of how to filter data for each tenant. In the system model, as depicted in Figure 13, components are aware of multi-tenancy and directly access a single database to query for the data necessary to complete requests. When querying the data it is the responsibility of each component to only query data related to the requested tenant or filter data while processing, to get results only for the current tenant.

As a result of using a single database for all tenants, the other components need to be aware of the context in which

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

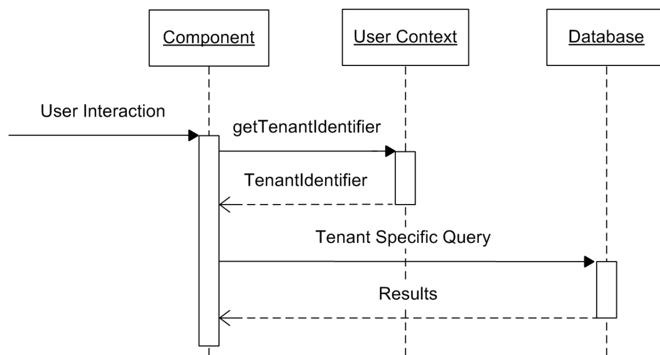


Fig. 13: Custom Property Object Pattern:Sequence Diagram

they operate. When retrieving data the components need to filter the results to only show data for the current tenant. The resulting interaction from component to database is depicted in Figure 13. The component first determines which tenant is currently active, this is done by using the User Context. It is implementation dependant how this User Context determines this, the only requirement is that it is able to tell a component which tenant is to be used in the context of the current request. The component then generates a query that is specific to the current tenant and sends this to the database. It is the responsibility of the component to ensure that the generated query only accesses data for the current tenant and to avoid retrieving data outside of tenant boundaries.

### D. Pattern Comparison

*1) Security:* Comparing the different data storage structures of the DATASOURCE ROUTER PATTERN and the CUSTOM PROPERTY OBJECT PATTERN shows that the DATASOURCE ROUTER PATTERN separates data from each tenant in a separate schema or database. This separation also guarantees that when a query is executed it will only return data for a single tenant without extra efforts from the developer. Because the *datasource router* component is the only component involved in selecting the datasource for a query, the changes of accidentally mixing data from multiple tenants due to programming errors are low. Failing to select a datasource would simply crash the application instead of mixing data from other tenants.

On the other hand, the CUSTOM PROPERTY OBJECTS PATTERN relies on the developers to write queries to only return data from the appropriate tenant. When no precautions are taken in the development and testing process the possibility of accidentally mixing data from multiple tenants is higher than when the DATASOURCE ROUTER PATTERN is used. When a correct filter is not applied in this pattern, users will receiving data from other tenants that should never be visible to them. When implementing this pattern it is critical to implement a strong test and quality assurance system as well as methods for automatically detecting queries that fail to filter data correctly.

At the system level the DATASOURCE ROUTER PATTERN requires a separate database or schema per tenant, these separate instances must all be monitored, updated and secured separately. Automation of security related system administration tasks is important, to ensure that all instances are always in the required state. Failing to implement proper procedures might result in tenant instances being in different states of updates and security related configuration settings. Security procedures for the custom property objects pattern can be simpler, because only a single database needs to be monitored and secured. This single database system is however a more high value target from a security perspective because data from all tenants is stored in a single place.

*2) Performance:* The CUSTOM PROPERTY OBJECTS PATTERN uses only a single large database or schema, which allows the database server to allocate all resources to one entity. The DATASOURCE ROUTER PATTERN requires a separate database or schema for each tenant, which, depending on the database system used, can result in partitioning of available resources like memory and caches and requiring more network resources to connect to all databases separately. Query efficiency in the custom property objects pattern is dependent upon the design of the database schema.

If the schema is generic, storing all data in field types without type information, the database engine will not be able to apply optimizations for specific datatypes. For example, storing fixed length integers in a variable length BLOB-field does not allow the database engine to make use of the known length of the field for faster searching through the storage structures. Designing the schema to partition data by tenant allows the database to limit the amount of data that is necessary to retrieve when executing a query for a single tenant. This limitation comes naturally for the DATASOURCE ROUTER PATTERN, because the data for each tenant is stored separately.

*3) Scalability:* Two types of scalability exist; vertical scalability and horizontal scalability. In vertical scalability we consider the amount of added capacity available when increasing the resources of a single system, e.g., adding more memory, more storage or more processing power to a single server. This is naturally limited by the available hardware options and associated costs of those components. Horizontal scalability concerns the scalability of adding more instances instead of increasing capacity in a single system. Horizontal scalability does not have the implied limits of available hardware that exist in vertical scalability, however, achieving perfect horizontal scalability has several challenges in coordination of nodes in a system. In practice this coordination costs resources, which makes it hard to achieve linear scalability in systems that require coordination of their workload.

By applying the CUSTOM PROPERTY OBJECTS PATTERN the application will only use a single database system. This impacts scalability in the application that requires a database system that is able to scale by itself to achieve scalability of the system as a whole. For example, a database system that supports clustering is appropriate to support scalability of the custom property objects pattern. In the DATASOURCE ROUTER PATTERN adding additional sources by moving part of the databases to separate servers is possible and does not require a database system capable of clustering.

The DATASOURCE ROUTER PATTERN is easier to scale out when the amount of tenants increases. An example case is a system currently using two database systems. In this example system, new tenants subscribe to the service and the

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

TABLE II: OVERVIEW OF BOTH DYNAMIC DATAMODEL EXTENSION PATTERNS

| | Datasource Router Pattern | Custom Property Object Pattern |
|---|---|---|
| Security | + Natural separation of datasets<br>+ Single point of selecting correct datasource<br>- More datasources to secure and maintain | + Only a single datasource to secure and maintain<br>- Risk of losing data separation with programming errors |
| Performance | + Correct data-types allow for optimizations<br>- Resource partitioning across separate schemas | + Full resource utilization across all schemas<br>- Loss of optimizations due to lack of type information |
| Scalability | + Natural scalability due to separate schemas<br>+ No need for scalability support in database | - No inherent scalability in pattern structure<br>- Requires database system capable of scaling |
| Maintainability | - Large number of possible database schemas must be tested<br>- Problem solving requires schema variants to be included | + Single static database schema<br>+ Custom properties can be handled with generic shared code |
| Implementation Effort | + Central component to handle all data-access<br>- Custom properties must be handled in all components | - Requires adaption of data-access in all components<br>- Custom properties must be handled in all components |

capacity becomes insufficient to service all tenants. Horizontal scalability is possible by adding two more database systems, effectively doubling the database capacity by allowing the data for new tenants to be stored on the new systems. There is virtually no overhead involved in this addition, because no extra coordination is required between the database systems servicing data for separate tenants.

The CUSTOM PROPERTY OBJECTS PATTERN requires a database system that is able to store all data for all tenants. The database system must in that case support vertical scalability by increasing the capacity of a single system instead of horizontal scalability. The application of a database system that provides a scalability capability is necessary for large deployments of this pattern. The results are dependant upon the effectiveness with which the database system deals with scalability challenges.

*4) Maintainability:* When extending the application with new functionality both patterns require that the new functionality is aware of any customized objects. For the DATASOURCE ROUTER PATTERN this involves creating a solution able of determining all database schema variations and correctly copying these values. The code involved can be complex because of the need to support various database modifications supported by the underlying database system. In the CUSTOM PROPERTY OBJECTS PATTERN, the extra properties are stored as predefined database objects, which can be handled the same as any other object stored in the database of the application. This means the code to handle the custom properties can be much simpler. A generic system could always handle the custom properties in the same way agnostic of their contents because they are abstracted as normal database objects. For problem solving a similar difference exists.

A problem affecting a single tenant in an application using the DATASOURCE ROUTER PATTERN can be harder to reproduce because of the various schema changes that could be done to the schema for that specific tenant. Because the changes, it is harder to isolate the root-cause of the problem. The CUSTOM PROPERTY OBJECTS PATTERN deals with a fully standardized database schema where the possible types of custom properties are explicitly visible in the design of the system. Because of this it is easier to create correct test-cases for the CUSTOM PROPERTY OBJECTS PATTERN, whereas the DATASOURCE ROUTER PATTERN has much more potential schema-variations, which must be explicitly handled correctly and tested.

*5) Implementation Effort:* For the DATASOURCE ROUTER PATTERN the initial implementation requires the development of the router component as well as systems to manage and automatically deploy new database instances for new tenants. The other components can however be left unchanged because awareness of the multi-tenant environment is not required. Using the CUSTOM PROPERTY OBJECTS PATTERN, on the other hand, does not require the development of new components or management systems. For this pattern the existing components need to be adapted to query the right data and use appropriate filtering methods. Both patterns require the implementation of code handling the existence of custom properties for entities in the applications data model. This is equal for both patterns and thus of no influence in a comparison on implementation effort.

## VII. CONCLUSION

Within this paper two problem domains related to implementing runtime variability in online business software are discussed. Also a pattern description method is proposed, suggestion the use of the following description levels: 1) Functional level, 2) System level and 3) Implementation level.

First, two dynamic functionality adaptation patterns, which are the COMPONENT INTERCEPTOR PATTERN and the EVENT DISTRIBUTION PATTERN are compared in terms of security, performance, scalability, maintainability and implementation effort. Both patterns offer a solution for dynamically adapting functionality of an online software product, but do so in different ways. The COMPONENT INTERCEPTOR PATTERN performs less in terms of *scalability*, because the interceptors can not scale independently of the application. When scaling up in terms of number of servers, the interceptors need to be available to all servers. Related to this issue, the *maintainability* of the COMPONENT INTERCEPTOR PATTERN is also less than that of the EVENT DISTRIBUTION PATTERN. This is caused by the fact the interceptors can not be decoupled from the rest of the system, creating a software product that will be difficult to maintain. The EVENT DISTRIBUTION PATTERN offers more isolation in terms of *security* than the other pattern, but requires more processing and network resources in terms of *performance*. Related to *implementation effort*, the COMPONENT INTERCEPTOR PATTERN is easier to implement, because no message broker or related services are required.

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

In general, the COMPONENT INTERCEPTOR PATTERN serves best for adapting functionality of small projects, where the EVENT DISTRIBUTION PATTERN is better for large projects, considering the quality attributes described in this paper.

Second, two dynamic data model extension patterns, being the DATASOURCE ROUTER PATTERN and CUSTOM PROPERTY OBJECT PATTERN are presented and evaluated. We conclude that the DATASOURCE ROUTER PATTERN has advantages on *security* by naturally isolating the data for all tenants, *scalability* by allowing for the distribution of tenants across datasources and *implementation* by not requiring all queries and components to be adapted but providing a single router component instead. The custom property objects pattern holds an advantage on performance by allowing better resource utilization, however, extra care is necessary to design an appropriate database schema. The CUSTOM PROPERTY OBJECTS PATTERN also scores better on *maintainability* by allowing standardized handling of the dynamic properties and using a static data model avoiding the need to test every possible variation when adapting the software.

For future work we are currently setting up larger evaluation sessions in which different patterns will be evaluated using experts. The evaluation of patterns is particularly difficult, because you should evaluate an abstract solution instead of a specific implementation. We are working on a structured method for comparing sets of patterns and making use of the implicit knowledge of experts. By doing this, we aim at evaluating the *solution*, instead of just an *implementation*.

## ACKNOWLEDGMENT

The authors would like to thank Allard Buijze and Koen Bos for helping in reviewing the results of the research.

## REFERENCES


[1] J. Kabbedijk, T. Salfischberger, and S. Jansen, "Comparing two architectural patterns for dynamically adapting functionality in online software products - best paper award," in *Proceedings of the 5th International Conferences on Pervasive Patterns and Applications (PATTERNS 2013)*, 2013, pp. 20–25.

[2] A. Dubey and D. Wagle, "Delivering software as a service," *The McKinsey Quarterly*, vol. 6, pp. 1–12, 2007.

[3] S. Aulbach, T. Grust, D. Jacobs, A. Kemper, and J. Rittinger, "Multi-tenant databases for software as a service: schema-mapping techniques," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM, 2008, pp. 1195–1206.

[4] S. Jansen, G. Houben, and S. Brinkkemper, "Customization realization in multi-tenant web applications: case studies from the library sector," *Web Engineering*, pp. 445–459, 2010.

[5] K. Pohl, G. Böckle, and F. van der Linden, *Software product line engineering: foundations, principles, and techniques*. Springer-Verlag, New York, 2005.

[6] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel, *A pattern language*. Oxford University Press, 1977.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Addison-wesley Reading, MA, 1995, vol. 206.

[8] P. Evitts and D. Hinchcliffe, *A UML pattern language*. Macmillan Technical Publishing, 2000, vol. 201.

[9] D. Mapelsden, J. Hosking, and J. Grundy, "Design pattern modelling and instantiation using dpml," in *Proceedings of the 40th International Conference on Tools Pacific: Objects for internet, mobile and embedded applications*. Australian Computer Society, Inc., 2002, pp. 3–11.

[10] D. Maplesden, J. G. Hosking, and J. C. Grundy, "A visual language for design pattern modelling and instantiation." in *Design Pattern Formalization Techniques*, 2007, pp. 338–339.

[11] A. Lauder and S. Kent, "Precise visual specification of design patterns," in *ECOOP98Object-Oriented Programming*. Springer, 1998, pp. 114–134.

[12] M. Jaring and J. Bosch, "Representing variability in software product lines: A case study," *Software Product Lines*, pp. 219–245, 2002.

[13] J. Bayer, . Gerard, O. Haugen, J. Mansell, B. Møller-Pedersen, J. Oldevik, P. Tessier, J. Thibault, and T. Widen, "Consolidated product line variability modeling," in *Software Product Lines*. Springer, 2006, pp. 195–241.

[14] R. Mietzner, T. Unger, R. Titze, and F. Leymann, "Combining Different Multi-tenancy Patterns in Service-Oriented Applications," in *Proceedings of the IEEE International Enterprise Distributed Object Computing Conference*, 2009, pp. 131–140.

[15] J. Kabbedijk and S. Jansen, "The role of variability patterns in multi-tenant business software," in *Proceedings of the WICSA/ECSA 2012 Companion Volume*. ACM, 2012, pp. 143–146.

[16] M. Svahnberg, J. van Gurp, and J. Bosch, "A taxonomy of variability realization techniques," *Software: Practice and Experience*, vol. 35, no. 8, pp. 705–754, 2005.

[17] A. Benlian and T. Hess, "Opportunities and risks of software-as-a-service: Findings from a survey of it executives," *Decision Support Systems*, vol. 52, no. 1, pp. 232–246, 2011.

[18] A. Hevner and S. Chatterjee, *Design research in information systems: theory and practice*. Springer, 2010, vol. 22.

[19] P. Runeson and M. H'ost, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, vol. 14, no. 2, pp. 131–164, 2009.

[20] J. Rumbaugh, I. Jacobson, and G. Booch, *Unified Modeling Language Reference Manual, The*. Pearson Higher Education, 2004.

[21] W. Van der Aalst, A. ter Hofstede, and M. Weske, "Business process management: A survey," *Business Process Management*, pp. 1019–1019, 2003.

[22] B. Carpenter, G. Fox, S. Ko, and S. Lim, "Object serialization for marshalling data in a java interface to mpi," in *Proceedings of the ACM 1999 conference on Java Grande*. ACM, 1999, pp. 66–71.

[23] M. Fowler, *Patterns of enterprise application architecture*. Addison-Wesley Professional, 2003.

[24] R. Weinreich, T. Ziebermayr, and D. Draheim, "A versioning model for enterprise services," in *Advanced Information Networking and Applications Workshops, 2007, AINAW'07. 21st International Conference on*, vol. 2. IEEE, 2007, pp. 570–575.

[25] W. Sun, X. Zhang, C. Guo, P. Sun, and H. Su, "Software as a service: Configuration and customization perspectives," in *Congress on Services Part II*. IEEE, 2008, pp. 18–25.


2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org