Comparing Two Architectural Patterns for Dynamically Adapting Functionality in Online Software Products

J. Kabbedijk, T. Salfischberger, S. Jansen
Department of Information and Computing Sciences
Utrecht University, The Netherlands
J.Kabbedijk@uu.nl, Tomas@salfischberger.nl, Slinger.Jansen@uu.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 that solve the problem of dynamically adapting functionality of an online software product; the Component Interceptor Pattern and the Event Distribution 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 is best suited for small projects, making the Event Distribution Pattern best for large projects.**

Keywords*—*architectural patterns; quality attributes; software architecture; variability;

I. Introduction

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 [1]. 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 [2]. One of the challenges in multi-tenant application architectures is the implementation of tenant-specific requirements [3]. Variability of software to fit requirements of specific customers can no longer be done by directly modifying the application for each client, because a single application serves multiple customers. 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.

The concepts of variability and quality attributes are explained in section II, after which the expert evaluation used is explained in section III. The architectural problem related to variability, faced by software architects, is explained in section IV. 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. The patters are compared in terms of security, performance, scalability, maintainability and implementation effort, of which the results in be found in a summarizing table in section VII.

II. Related Work

Variability - The field of software variability has been the subject of research from both the modeling perspective as well as the technical perspective. Software variability modeling is common in software product lines as described by Jaring and Bosch [4]. The application of variability modeling as used in product line variability [5] to software as a service environments has been described by Mietzner, Metzger, Leymann, and Pohl [6]. Variability modeling as dicussed in the aforementioned works contributes to the understanding of where the application architecture needs to be able to accomodate change or extension. Patterns play an important role in modeling and solving variability in software products [7].

Svahnberg, van Gurp, and Bosch [8] propose feature diagrams as a modeling technique to describe the different variants of feature in a software product. Svahnberg et al [8] 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. [8] 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 [9] 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 as found by Benlian and Hess [9]; cost. 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 [10] 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. Solutions that are observed in multiple at least three products are presented as patterns and are evaluated by two domain experts as feedback mechanism. The evaluation of the cases by experts enhances the validity of the cases, as described by Runeson and Höst [11].

During each evaluation session, a pattern is 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. Architectural Problem Definition

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 [12]. 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 is needed that allow to configure this functionality. This functional situation is modeled in figure 1, 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*.

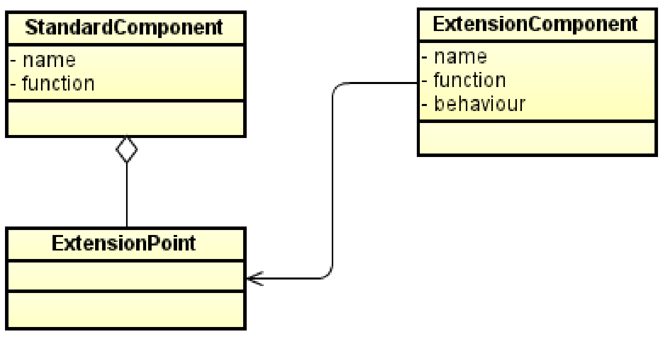

Fig. 1. Functional Model for adapting functionality

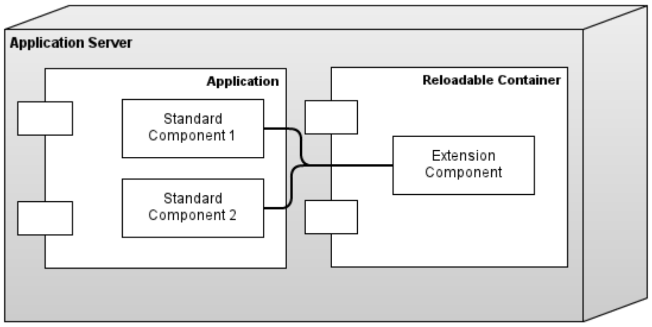

Fig. 2. Component Interceptor Pattern: System Model

V. Dynamical Functionality Adaptation Patterns

This section presents two different patterns, both offering a solution to dynamically adding functionality to a software product.

Component Interceptor Pattern - The Component Interceptor Pattern as depicted in figure 2 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 [13].

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

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.

Event Distribution Pattern - The system model in figure 4 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 5 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

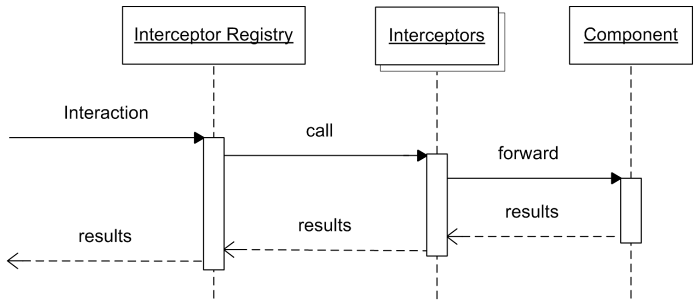

Fig. 3. Component Interceptor Pattern: Sequence Diagram

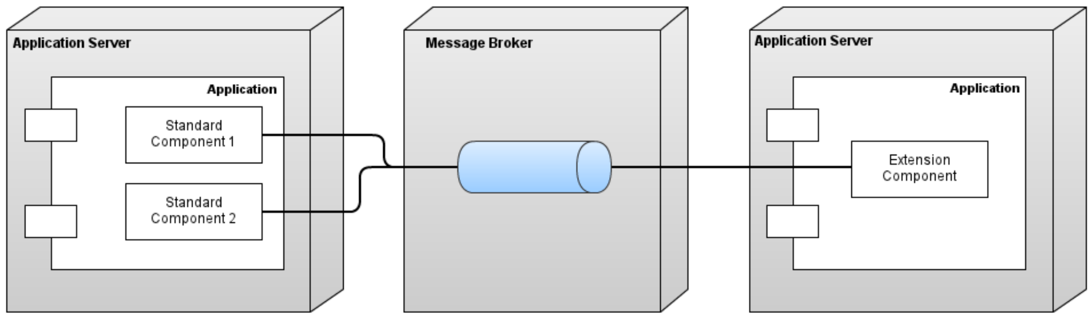

Fig. 4. Event Distribution Pattern: System Model

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. If on the other hand 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*.

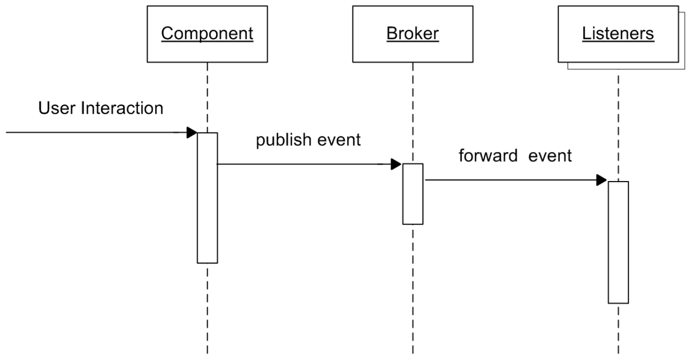

Fig. 5. Event Distribution Pattern: Sequence Diagram

VI. Pattern Comparison

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

A. 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 2. 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. This isolation would however have an impact on performance because of the nature of the integration.

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 3. 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 [14].

B. 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. For security reasons it could however 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.

C. 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.

The EVENT DISTRIBUTION PATTERN on the other hand 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, Cloverleaf and others.

D. 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 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 [15].

E. 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.

VII. Conclusion

Within this paper 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, both 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 which 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. 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.

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. 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 shoud 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 evaluation 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] A. Dubey and D. Wagle, "Delivering software as a service," *The McKinsey Quarterly*, vol. 6, 2007.

[2] 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 ACM SIGMOD international conference on Management of Data*. ACM, 2008, pp. 1195–1206.

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

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

[5] J. Bayer, S. Gérard, O. Haugen, J. Mansell, B. Moller-Pedersen, J. Oldevik, P. Tessier, J. Thibault, and T. Widen, "Consolidated product line variability modeling," 2006.

[6] R. Mietzner, A. Metzger, F. Leymann, and K. Pohl, "Variability modeling to support customization and deployment of multi-tenant-aware software as a service applications," in *Proceedings of the ICSE Workshop on Principles of Engineering Service Oriented Systems*. IEEE Computer Society, 2009, pp. 18–25.

[7] J. Kabbedijk and S. Jansen, "The role of variability patterns in multi-tenant business software," in *Proceedings of the Joint 10th Working IEEE/IFIP Conference on Software Architecture and 6th European Conference on Software Architecture Companion*. ACM, 2012, pp. 143–146.

[8] 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.

[9] 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.

[10] A. Hevner and S. Chatterjee, "Design science research in information systems," *Design Research in Information Systems*, pp. 9–22, 2010.

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

[12] W. Van der Aalst, A. ter Hofstede, and M. Weske, "Business process management: A survey," *Business Process Management*, pp. 1–12, 2003.

[13] 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 Conference on Java Grande*. ACM, 1999, pp. 66–71.

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

[15] R. Weinreich, T. Ziebermayr, and D. Draheim, "A versioning model for enterprise services," in *21st International Conference on Advanced Information Networking and Applications Workshops*, vol. 2. IEEE, 2007, pp. 570–575.

TABLE I. OVERVIEW OF BOTH DYNAMICAL FUNCTIONALITY ADAPTATION 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. - The broker system requires extra resources.
Scalability	- No independent scaling of extension components. - Does not scale to high number of extension components.	+ Independent scaling of extension components. + Extension components cannot delay standard components. - Requires scalable message-broker system.
Maintainability	- Tight coupling of extension components.	+ Loose coupling of extension components.
Implementation Effort	+ Direct communication with standard components. + Access to all data by design.	- Requires the setup of a message broker system. - Requires a separate mechanism to communicate with the application.