Design Patterns for Runtime Variability of Business Software as a Service

Tomas Salfischberger
3143945 – tjsalfis@cs.uu.nl

Contents

Chapter 1

Introduction

In recent years Software as a Service (SaaS) had proven to be a highly popular deployment model for multi-tenant software. The multi-tenant software architectures used to accomodate a SaaS deployment model provide some unique challenges in separation of different users and scalability to a high number of users. For example the datamodel of such an application requires special architecture principles to accomodate data for multiple tenants (Aulbach, Grust, Jacobs, Kemper, & Rittinger, 2008).

Another challenge in multi-tenant application architectures is the implementation of requirements that are specific to a single client (Jansen, Houben, & Brinkkemper, 2010). Due to the shared nature of the application it is no longer possible to modify the code to meet a specific clients requirements without affecting other clients. To be able to meet customer requirements the application needs to support runtime variability.

In this research we gather and analyze a set of software design patterns to add support for runtime variability to multi-tenant application architectures. The patterns vary in their impact on the technical properties of the software like performance and maintainability, their impact on the cost-drivers of the SaaS business model and the requirements they can fulfill.

1.1 Relevance

The Software as a Service model has a clear set of advantages to both the software vendor and the users. 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 (Dubey & Wagle, 2007). In the SaaS deployment model a single application serves a large number of customers. Because of this the development and setup of the application can be amortized over all contracts, resulting in lower total cost for the software.

The lack of variability is a hurdle keeping vendors and their clients from switching from traditional software deployments to SaaS. With this research we intend to provide organisations interested in adopting variability in their SaaS offering with examples of common patterns as well as a method to evaluate these patterns before implementation.

We will also provide a method for describing and documenting variability patterns. These methods can be used by others to describe additional patterns for different areas of variability and evaluate those patterns by the criteria we gathered. A full catalog of patterns can be made availabe to practitioners interested in implementing variability in a multi-tenant software product.

1.2 Key concepts

Software as a Service is a relatively new business model, because of this the terminology used to describe software using a SaaS business model and the necessary multi-tenant application architecture is used differently by various sources. To avoid confusion we have identified a set of key concepts and the relations between these concepts. Considering a single software product the concepts involved are depicted in figure 1.1. We use the term software product to describe a piece of software, its supporting systems and all auxiliary materials like documentation offered by a vendor to their customers. The customers are considered tenants, because the customer does not actually purchase a copy of the software, instead the software is rented from the vendor. For a software product we consider a set of variability points, linked to a specific variability pattern used to implement this point. Although the user of the software product is important in determining variability requirements, we consider the tenant as the granularity at which variations are implemented. For certain applications a tenant could be a single private user in the case of consumer oriented software (e.g. Google's Gmail), however for business to business applications a tenant is generally considered to be a complete organisation or business unit.

Variability can be implemented in various ways, each with different influences on performance, maintainability and security. The patterns we evaluate describe generic implementation approaches to variability. We consider each customization that is necessary to satisfy the variability requirements of the tenant separately. These customizations are implemented using a variability realization technique (VRT).

The patterns we describe in the research are patterns for the implementation of variability realization techniques. Each pattern describes a single technique for implementing a specific type of variability. We limit the gathered patterns in this research to two areas of variability, however the methods and concepts are applicable to other areas as well.

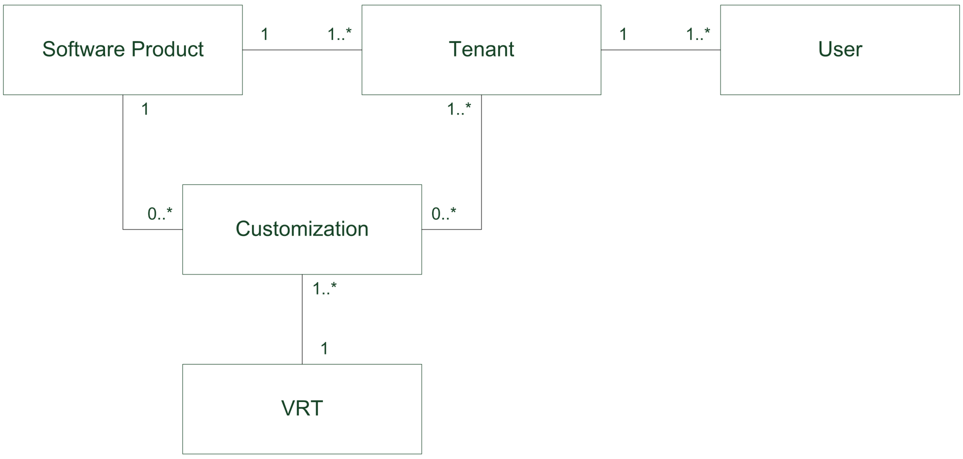

Figure 1.1: The key concepts involved in customizable multi-tenant software

1.3 Thesis Structure

Chapter 1 introduces the subject and key concepts. Following this introduction we describe our research questions and method in chapter 2. Chapter 3 describes the context of our research in relation to research in the fields of architecture description languages, software patterns, evaluation methods and software variability. We conduct a literature study to determine the evaluation criteria and description languages for the selected patterns in chapters 4 and 5, evaluating description methods for patterns and criteria for the evaluation of patterns respectively.

The second part of this thesis focusses on two variability areas and the applicable patterns in chapters 6 and 7. Chapter 6 describes patterns for variability on the datamodel level of the applicaion, chapter 7 applies to patterns for varying behaviour of the application. For the description of those patterns we apply the method defined in chapter 4. In chapter 8 we analyze patterns using the criteria selected in the first part to provide readers interested in implementing variability in their software a comprehensive overview of the properties and pitfalls of the applicable patterns.

We finish with an expert review of our analysis results in chapter 9 and conclusion in chapter 10.

Chapter 2

Research Approach

2.1 Research Objectives

Custom software development is costly and often associated with budget overruns, late delivery and large business risks. When an off the shelf software product exists that meets the organisations requirements this is usually a more cost effective solution. In the real world the decision to build custom software or select an existing software product is not a simple choice between the two options. Existing software products might not fit the complete set of requirements, but offer options to extend the product at some points or to replace parts of the product. These extensions could then be built to meet the specific requirements of the customer while the rest of the system suffices to meet the more generic requirements.

These variations can be implemented in various ways like adapting the application source code and delivering a version of the application specific for the customer or extending the application using a predefined plug-in architecture. When the application is delivered using a SaaS business model with a multi-tenant architecture, modifying the application for earch customer is no longer a viable option because all customers share the same application.

The objective of this research is to evaluate patterns for customization realization techniques (CRTs) that can be applied by a software vendor to realize runtime variability in a software product in a multi-tenant environment.

2.2 Research Questions

To set a scope for this research and determine the approach of pattern evaluation we have defined the following research questions and subquestions:

1. How can software design patterns for runtime variability be evaluated in the context of a multi-tenant business application?

  a What description methods are applicable to software design patterns?
  b What are evaluation criteria for software design patterns in the context of business software?

This question is at the basis of the research method and comparison framework. Determining the description languages available for patterns and selecting a method to describe patterns allows us to make different software design patterns comparable. The context definition further scopes this research and sets a reference for definitions in this research. The selection of evaluation criteria forms the basis for the second part of the research in which we evaluate selected patterns by the following questions:

2. How can a multi-tenant business application provide runtime variability on datamodel and application behaviour?
    a What software design patterns are applicable to runtime variability of the datamodel?
    b What software design patterns are applicable to runtime variability of application behaviour?

The patterns selected in this question and its subquestions form the core of the research. We will describe the patterns using the description methods selected in research question 1 a, we then conclude with question 3:

3. How do the selected patterns compare on evaluation criteria as selected in research qeuestion 1?

We evaluate the patterns selected and described in research qeustion 2 according to the methods and evaluation critera selected in research question 1 b. This results in a set of patterns applicable to each of the two types of variability, compared on their relevant evaluation criteria.

2.3 Research Method

The research approach is derived from the methods for design research as described by Vaishnavi & Kuechler (2007). Vaishnavi & Kuechler (2007) define design research to "involve the analysis of the use and performance of designed artifacts to understand, explain and very frequently to improve the behavior of aspects of Information Systems". The artifacts in this research are descriptions of implementation patterns for software variability combined with their respective properties on selected evaluation criteria. The goal of these descriptions is to improve the variability aspect of multi-tenant information systems. We have mapped the phases of this research on the

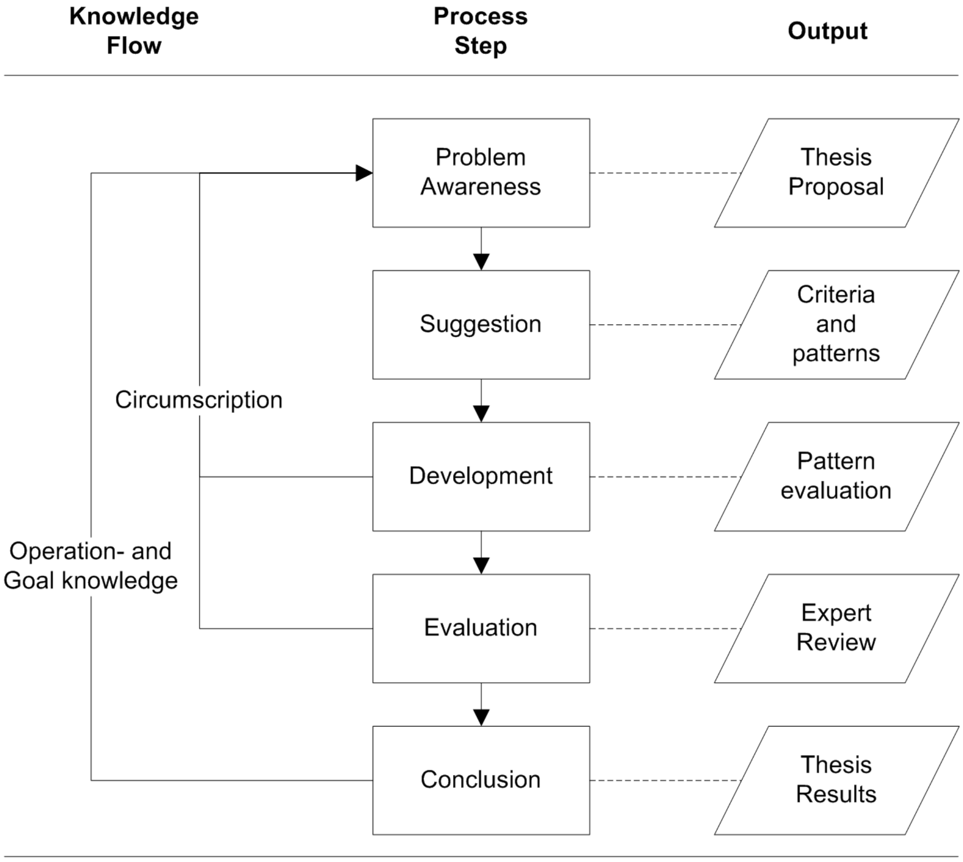

Figure 2.1: Mapping research phases and output to the design research method by Vaishnavi & Kuechler (2007).

Research Question	Research Approach
1.a	Literature study
1.b	Literature study
2.a	Case studies
2.b	Case studies
3	Design science

Table 2.1: Research approach per research question

steps defined for design research as depicted in figure 2.1 to provide insight in the research process.

First awareness of the problem is necessary to define the scope of the research and formulate a set of research questions that lead to a solution to the defined problem. The problem description, scientific contribution and social relevance are the inputs to the thesis proposal that is the starting point of this research.

Based on a literature study a suggestion for the problem solution is developed. We describe the context, evaluation criteria and patterns as the first step to a preliminary solution to the variability problem. The patterns are at this point not yet evaluated, so this preliminary solution has to be extended in the development step.

The development step applies the selected evaluation methods to the patterns to determine their applicability in different situations and relative performance characteristics on different measurements. This results in a description of two patterns for each variability type, combined with the results of the evaluation of each pattern.

To validate the assessment of the performance characteristics of the patterns and their descriptions we will conduct expert reviews in the evaluation phase of the research. Experts will be asked to validate our description and analysis of the patterns.

The final phase of the research is the conclusion phase in which we finalize the pattern overview and present the thesis results.

2.3.1 Approach per research question

The research questions map to the phases of the research. Question 1 sets the context of the research, questions 2a to 2c require gathering of the patterns and pattern descriptions using case-studies at SaaS-vendors matching our definitions from question 1. Question 3 applies the methods selected in 1b to the patterns described in question 2a to 2c. Each question requires a different research approach as described in table 2.1.

2.4 Validity

Validity is a measure for the accuracy with which a research reflects the concept that the researcher is attempting to create or measure (Yin, 2002). Yin describes four types of validity:

1. Construct validity
2. Internal validity
3. External validity
4. Reliability

2.4.1 Construct validity

Construct validity refers to the level of certainty with which an operational measurement accureatly measures the concept as it was intended to be studied. Construct validity can be an issue especially in case study research due to potential observer subjectivity (Tellis, 1997). Yin (2002) proposed multiple approaches to counteract this: using multiple sources of evidence, establishing a chain of evidence, and having a draft case study report reviewed by key informants.

For our case studies we use multiple cases which cover the same problems and take equal approaches to solving these problems, from these multiple cases we extract our patterns. Using multiple cases as the source of our patterns also ensures that patterns are generically applicable and not specifically tailored to a single software system or implementation environment.

The patterns and their evaluation are validated in an expert review by software architects. In this way we validate that all essentials parts of a pattern are captured and conclusions on the merits or problems with a pattern are validated against the practical experiences of the experts. Validation is done by experts in the field of software architectures, selected from the contacts of the authors.

2.4.2 Internal validity

Internal validity is concerned mainly with correct causal inferences. Yin (2002) argues that internal validity is therefore less important in the case of descriptive and explanatory research. Factors like controllability, history, maturation, testing, instrumentation, statistical regression and mortality as defined by (Cook, 1979) are only applicable to statistical research.

A possible problem with internal validity in our case study approach is (unconsciously) selecting only specific companies of case studies. To mitigate this problem we use random selection to select cases from the network of the supervisors.

2.4.3 External validity

External validity is based on generalization of the conclusions from a study to a larger population. It is thus important to make sure the results of case studies are general results and not specific to a single case. A risk to external validity is in the number of case studies, researching a small population makes it harder to guarantee that the results can be generalized to the full population.

The software design patterns we intend to capture are representations of common approaches in the industry. Each pattern is thus based on multiple input cases and represented in the generic form. Representing in a standard form contributes to generalizability of the pattern.

2.4.4 Reliability

Reliability refers to the repeatability of a study and the degree of consistency between repeated results. By describing the research methods and definitions of measurements we apply to the patterns it is possible for other researches to apply the same assessment criteria to the patterns. The results are dependant upon the expers interviewed and the experts that review the results. To increase reliability of the research we use experts from different organisations with backgrounds in different technologies, thus subjecting the results to a review from multiple different angles.

Chapter 3

Related work

3.1 Software 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 & Bosch (2002). Jaring & Bosch describes a case study at a software company specializing in product and system development for a professional mobile communication infrastructure. The case study found the need for explicit modeling of variability in a software architecture. Jaring & Bosch proposes a method to model variability in industrial software systems. Variability modeling has also been discussed in the context of product line variability by Bayer, Gerard, Haugen, Mansell, Mller-Pedersen, Oldevik, Tessier, Thibault, & Widen (2006), providing a metamodel for product line variability.

The application of the variability modeling techniques used in product line variability in a software as a service environment has been described by Mietzner, Metzger, Leymann, & Pohl (2009). 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. Mietzner et al. (2009) discuss the difference between Customer-driven variability and Realization-driven variability, which are discussed as external- and internal-variability in the context of product line variability modeling.

Svahnberg, van Gurp, & Bosch (2005) propose feature diagrams as a modeling technique to describe the different variants of feature in a software product. An example of a feature diagram is included in figure 3.1. Svahnberg et al. (2005) use there 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. (2005) describe implementation techniques

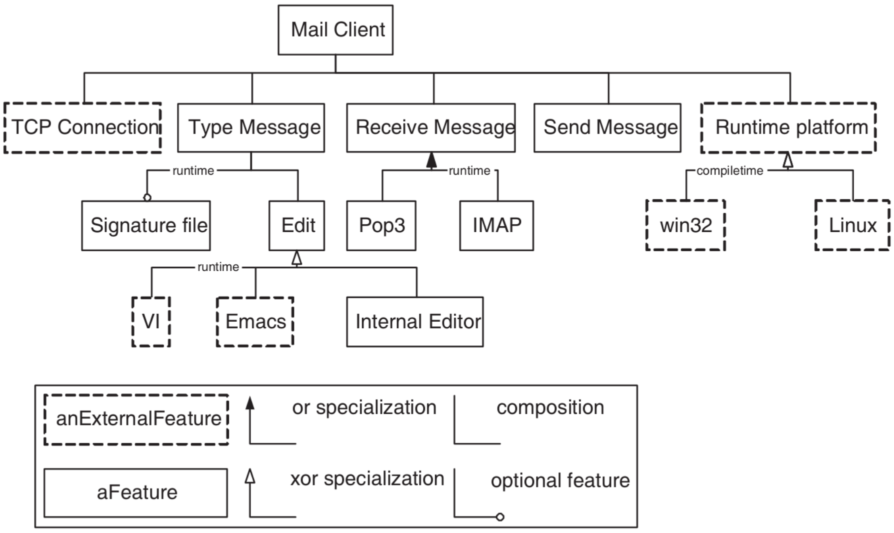

Figure 3.1: Example of a feature diagram as proposed by Svahnberg et al. (2005)

for variability per installation instance of the software whereas we focus on runtime variability in a multi-tenant context. Runtime variability is required in the context of multi-tenant SaaS, because the software cannot be re-built and re-configured to implement the variability requirements of a single tenant, because such a process would interrupt service to all tenants.

The importance of runtime variability is described by Montero, Pena, & Ruiz-Cortes (2008). The research by Montero et al. describes the differences between variability in a software product line as described by Svahnberg et al. (2005) and runtime variability. In software product line variability a static variant is created by selecting a set of standard components combined with a selected set of variants. The research compares the methods of Gomaa & Hussein (2007) with those of Svahnberg et al. (2005).

In the research of Gomaa & Hussein (2007) the concept of dynamic software variability is discussed. Combining software product lines based on reusable shared components with the ability to reconfigure the system at runtime. They propose a system based on a Reconfiguration Framework, which is able to reconfigure an application instance at runtime. The application is then transitioned from one runtime configuration to another. The reconfiguration step involves enabling and disabling architecture components of the application. While this allows the application to change at runtime it may not be suitable for software as a service applications with many tenants, because the reconfiguration is a separate process.

3.2 Software as a Service

Software as a Service is a delivery method in which software is not sold as a product but delivered as a service, commonly over the internet. Turner, Budgen, & Brereton (2003) describe software as a service as a demand-led software market in which businesses provide services when needed. Separating ownership of the software from its use by allowing users to subscribe to a service instead of buying the actual software. The envisioned market and setup of Turner et al. is mostly based on delivering these services as webservices using techniques like SOAP to allow clients to integrate all services into an application. This requires the client to develop a service integration layer which aggregates all the necessary services.

Dubey & Wagle (2007) describes software as a service more broadly as software delivered online and uses the example of a CRM and ERP system. The research focuses on the emerging trend of delivering software as a service and the threats this comprises for traditional software product organisations. Dubey & Wagle present data that underlines the growing investments in software as a service based businesses by venture capitalists as well as the decreased total costs of ownership (a slight misnomer for software as a service) for software as a service compared to traditional on premise software. They go one to review the revenue models of software as a service businesses, describing higher relative sales and marketing costs during the growth phase of the business. This is caused by the fact that sales and marketing costs are incurred completely before te sale, while the revenue of a software as a service subscription is only incurred over the following months or years.

Research by Marston, Li, Bandyopadhyay, Zhang, & Ghalsasi (2011) describes Software as a Service as the highest level of cloud computing. Comparing software as a services to lower levels of cloud computing like platform as a service or infrastructure as a service models. Marston et al. describe the under-utilization of on-premise IT resources as one of the factors in decreasing cost for service delivery models. They conclude that cost seems to be the main driver for replacing on-premise IT resources iwith resources deliverd as a service, while other benefits to the service delivery model exist. The main benefits listed are a decrease in up-front capital investment, lowering barriers to innovation, easier scalability and the possibility of completely new classes of applications which could not have been developed without the use of cloud computing.

3.3 Multi-tenancy

Multi-tenancy provides increased enconomies of scale and makes complex applications more accessible to small businesses (Dubey & Wagle, 2007). On this basis research has been conducted to determine the benefits of multi-tenancy compared to extra challenges involved. Bezemer & Zaidman (2010) discuss the added complexity of an architecture that supports multi-tenancy, considering the extra challenges on performance, security, scalability, uptime and maintenance. They describe the impact of multi-tenancy on authentication, security and the database, concluding that multi-tenancy could be favorable to maintenance as long as the multi-tenancy implementation is clearly confined within specific layers of the application architecture. This related to our research questions on maintainability impact of the variability patterns, we will consider the impact of proposed patterns on different layers of the architecture.

Guo, Sun, Huang, Wang, & Gao (2007) list the same complexities in multi-tenancy Bezemer & Zaidman found, they propose to isolate developers from the complexities in the isolation of tenants with respect to security, performance and availability by including these components in a generic application framework. By moving the complexities involved in these practices from the codebase to the framework they claim to decrease the maintenance complexities involved in multi-tenancy. Customization is discussed in the context of maintenance, where Guo et al. hilight the importance of on-the-fly customization to minimize application downtime. The approach to customization in the proposed framework is two-fold, first customization on a technical level should be minimized by providing simple settings for all basic types of customization (e.g. changing the interface colors or a logo image). The second step is to leverage the flexibility of a service oriented architecture by incorporating variability in BPEL processes which allow developers to attach extra steps to application workflows.

3.4 Software Patterns

Software patterns are reusable solutions to recurring problems in software development (Gamma, Helm, Johnson, & Vlissides, 1995). Gamma, Helm, Johnson, & Vlissides (1993) introdused the concept of object oriented design patterns with the purpose of creating a common vocabulary for design. Providing a library of the best solutions to common problems encountered in object oriented software design. Gamma et al. called design patterns re-usable micro architectures. The description of patterns for object oriented software design by Gamma et al. is widely known as one of the reference works for design patterns. Fowler extended upon the concept of object oriented design patterns with patterns for enterprise architectures (Fowler, 2002). Looking at patterns on a larger scale than just object oriented design, while still staying close to object orientation, describing patterins in UML.

Rosengard & Ursu (2004) describes patterns as proven solutions to recurring software construction problems in a given context. Considering the context and differences in levels, such as the object oriented design context used by Gamma et al. but also architectural patterns and language specific patterns. The research aims to develop tools for intelligent dissemination of patterns, allowing a wide audience to make use of the patterns. In contrast to other tools supporting software patterns Rosengard & Ursu provide tools that help developers implement patterns instead of replacing the developer by a pattern instantiated purely using software. The research recognizes pattern description methods ranging from informal descriptions, through UML to formal representations, finally proposing ontological representations of patterns. The ontological representation will, according to the authors, by standardizing pattern representations aid the development of automatic tools for organisation and retrieval of patterns.

3.5 Software Quality

The concept of software quality is important to this research, because the evaluation of patterns requires criterial on which patterns will be evaluated. However software quality has a different meaning to different audiences as described by Kitchenham & Pfleeger (1996). Kitchenham & Pfleeger describes quality from different views. The transcendental view, considering quality as something to strive for which is never achieved completely. The user view, evaluating quality in a task context, considering whether the users requirements were met. The manufacturing view, focussing on quality during production, considering the costs of reworks. The product view, considering the quality of software by its internal properties, such as software-metrics. Considering the different views it is according to Kitchenham & Pfleeger impossible to define a single measurement of quality. They conclude that it is important to determine the aspects of quality which are important in the current context before setting definitions and measurements to determine this quality.

ISO 9126 (International Organization for Standardisation, 1991) is a more rigored definition of software quality. The standard defines a quality model that consists of six characteristics of software quality and 27 sub-characteristics. For the sub-metrics the standard defines measurements to determine the level of quality software posesses on that sub-characteristic. The vales for all sub-characteristics together determine the quality on the main characteristic.

Jung, Kim, & shin Chung (2004) review the ISO 9126 standard (International Organization for Standardisation, 1991) to assert its merits as a universal standard of software quality. Jung et al. reviewed the internal consistency of the ISO 9126 quality attributes and sub-attributes by conducting surveys on users of software. The users were asked to rate a software application on the sub-characteristics of ISO 9126.The authors then analyzed the internal consistency between sub-characteristics of a single characteristic, expecting to find a high consistency. The analysis showed internal consistency for some characteristics while inconsistent sub-characteristics were found indicating that the placement of sub-characteristics in some characteristics may be arbitrary. It is important to note that only a single application was evaluated and the authors report that future research is necessary.

Another aspect of software quality is the cost of software defects. Slaughter, Harter, & Krishnan (1998) reviews the cost of software quality, describing the importance of quantifying the business-case of software quality improvement. Adding activities to the development process such as design reviews and code inspections adds to the development time of a product and might thus according to Slaughter et al. be seen as an unnecessary increase of costs. The authors propose to define a metric, return on software quality (ROSQ), calculating the ratio of cost saved on rework and quality problems to the investment made in quality improvement efforts. The research showed a diminishing return on subsequent quality improvement efforts, each next effort improved quality however the rate at which quality improved decreased with each step. Slaughter et al. conclude that software quality improvement effort should be considered an investment and organisations should strive to find the optimal ROSQ.

Chapter 4

Pattern description methods

4.1 Software Design Patterns

Object oriented design patterns were first introduced by Gamma et al. (1993) and later popularized by their book Design patterns: elements of reusable object-oriented software (Gamma et al., 1995). Gamma et al. define design patterns as recurring patterns of classes and communicating objects in many objectoriented systems. They state that: Each design pattern systematically names, explains, and evaluates an important and recurring design in objectoriented systems. We distinguish the patterns described in this research from the original object orientd 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..

Others have, based on the first set of design patterns by Gamma et al., researched the best methods for describing and communicating design patterns for later re-use. For example Evitts (2000) applies the UML to design patterns and proposes a modeling technique based on UML-modeling. The same approach is taken by Mapelsden, Hosking, & Grundy (2002) 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 that links the elements of design patterns in DPML to UML model elements. Mapelsden et al. 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 (Mapelsden, Hosking, & Grundy, 2007).

Lauder & Kent (1998) 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. Their main concern however 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.. 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 & Kent 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.

4.2 Design patterns in multi-tenant systems

The application of design patterns to the description of multi-tenant systems is different from a pure object oriented design pattern. 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. As discussed in our introduction we consider the software not only to be a set of source files, but to include supporting systems like a database or message-bus and related auxiliary materials like documentation. 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 & Kent. Instead of modeling different levels of detail and abstraction within only object oriented design we will model different levels of the software architecture including supporting systems. The levels we wil describe are:

1. Functional model
2. System model
3. Implementation diagrams

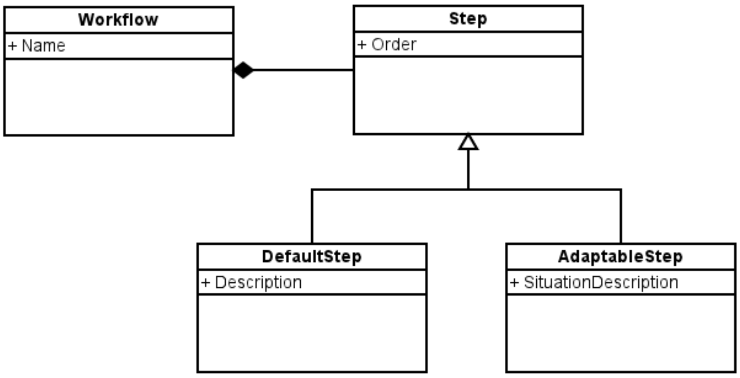

Figure 4.1: An example UML class diagram.

4.2.1 Functional model

The first level of modeling is the functional level. At this level we describe the patterns functional intention in a technical context, comparable to the type-model of Lauder & Kent. Multiple patterns can share the same model at the functional level, because multiple patterns can be designed to reach the same functional effect with different performance and scalability characteristics. For the graphical modeling of the functional level we propose the application of UML class diagrams. In this diagram we capture the functional situation that would result from application of the pattern without considering implementation of pattern instantiation details.

The basis of the functional model is the problem statement or requirement that is the trigger to implement this type of variability. In this research we limit the scope to two problem statements, Datamodel extesion and Adding behaviour (chapters 6 and 7). All patterns discussed in those problem areas thus share a common problem statement and functional model. This shared functional model asserts that a valid comparison of the patterns on their quality attributes is possible. When patterns don't share a functional model it will be hard to compare then without involving differences in the envisioned situation in the comparison.

The UML class diagram conveys the structural view of a software system, consisting of classes and interfaces and the relationships among them (Rumbaugh, Jacobson, & Booch, 1999). The application of UML class diagrams to software pattern description is common to multiple software pattern description languages, e.g. DPML (Mapelsden et al., 2002), UML pattern language (Evitts, 2000) and UML-Based Pattern Specification (France, Kim, Ghosh, & Song, 2004). An example of a UML class diagram is included in figure 4.1.

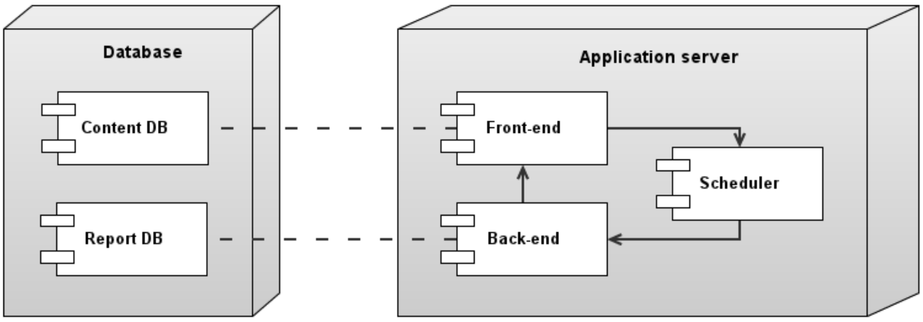

Figure 4.2: An example UML deployment diagram.

4.2.2 System model

The second level of modeling is the system model. At this level we model the overview of the software including supporting systems after the application of the pattern. This model shows interaction of different components within and between systems as a result of the implemented pattern. For graphical modeling we apply UML deployment diagrams (Rumbaugh et al., 1999). A UML deployment diagram is a nested diagram showing systems and their underlying components and interaction within the complete system. Figure 4.2 illustrates the nested levels in a UML deployment diagram.

The example in figure 4.2 shows parts of the system, a database and an application sever, each containing multiple components. The links depict interaction and dependencies between components, either within a part of the system or across boundaries. Using the system model we gain a better overview of which parts of the system are affected by implementing the pattern. This allows for more accurate estimations of the impact.

4.2.3 Implementation diagrams

The third level of modeling contains the implementation diagrams. These diagrams depict the spcific implementation of the components of the proposed pattern. The implementation diagrams are closely related to the system model, they depict in more detail the method of application of the components in the system model.

The modeling technique used for the implementation diagrams is dependent upon the area of application of the pattern and the context in which a component operates. Examples of modeling techniques to use at this level include ER-diagrams (Chen, 1976) as depicted in figure 4.3 for describing database structures related to the implementation of a pattern, UML class diagrams for describing the internal structure of a component or

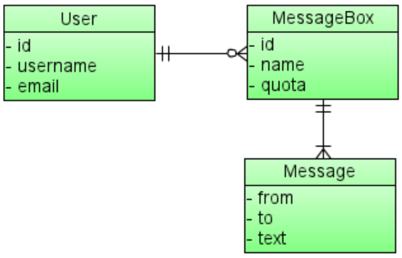

Figure 4.3: An example of an ER-diagram

interaction diagrams, for example a sequence diagram as depicted in figure 4.4, to describe the interaction between components in a pattern.

Care must be taken to make sure that the used implementation diagrams only depict the implementation parts that are relevant for the distinguishing features of a pattern. Behaviour or structure that is not related to this specific pattern should not be modeled to avoid confusion about the patterns intention.

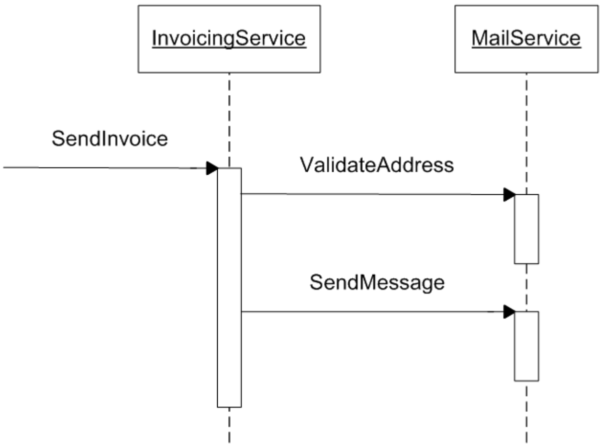

Figure 4.4: A sequence diagram to describe interaction between components.

Chapter 5

Pattern evaluation criteria

To determine the criteria for evaluating patterns in business software as a service we first need to define the boundaries of what we consider business software as a service. This question is two-fold, first, how do we define business software and second, how do we define software as a service (SaaS).

We define business software as software that is offered as a product for use in the context of a business. This is a very broad definition, generally including anything from desktop productivity suites to organizaton wide CRM or ERP systems. We specifically exclude software that is created for entertainment and home use.

The evaluation of software design patterns is two-fold; evaluating the pattern-description and evaluating the expected results of pattern instantiation. Evaluating the pattern itself by verifying properties like clarity of the description, adherence to object oriented design rules and if the pattern actually solves the proposed problem has been done by Hsueh, Chu, & Chu (2008). Their results show that a pattern can be evaluated on its internal properties, they propose a quantitative approach to measure the effectiveness of a pattern. In this research we evaluate the expected results of pattern instantiation. We will therefore determine desired quality attributes of the resulting architecture which can be affected by the patterns we evaluate.

Our evaluation is similar to the evaluation of software architectures, the same desired quality attributes are used and the reasoning is on the level of quality attributes of the whole system. A dissimilarity is the fact that we evaluate only the quality attributes that are to be affected by the selected patterns on a meta-level, without evaluating a concrete pattern instantiation in a software architecture. An overview of the relation between the different components is included in figure 5.1. We compare the effect on desired quality attributes in comparison to patterns sharing the same functional model. Generically comparing patterns with different functional models is not one of the goals of our evaluation method.

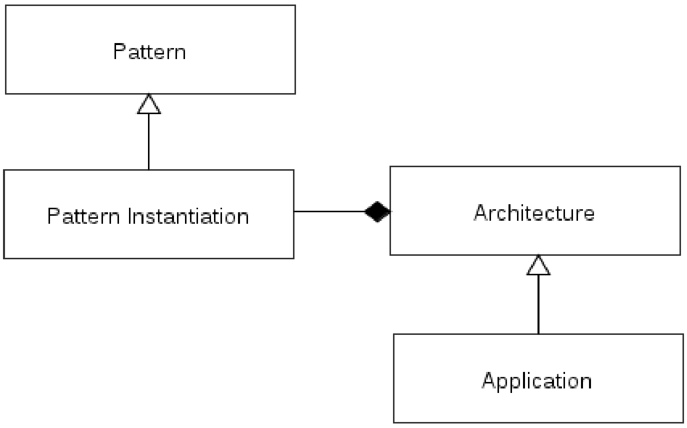

Figure 5.1: The relationship between pattern, pattern instantiation and architecture

5.1 Desired quality attributes in business SaaS

To compare patterns on their relative qualities we first need to determine which quality attributes are desirable in our evaluation context. We select these attributes broadly, it is up to the software engineer using the patterns to determine the importance of the selected quality attributes in the specific implementation context. At implementation time a method like the Architecture Tradeoff Analysis Method (Kazman, Klein, Barbacci, Longstaff, Lipson, & Carriere, 1998) can be used to determine the weight that must be given to each quality attribute.

Research has been conducted to find the risks and opportunities as perceived by organisations actively adopting SaaS and organisations that haven't (yet) adopted SaaS by Benlian & Hess (2011) and Wu, Lan, & Lee (2011). Benlian & Hess (2011) finds that security is 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 a risk perceived by SaaS users is closely related to the most important perceived opportunity as found by Benlian & Hess (2011), cost. When performance is insufficient, clients are lost, when the system uses too much resources to gain an acceptable level of performance, cost is increased (Espadas, Molina, Jimnez, Molina, Ramrez, & Concha, 2011). 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 more or more specialized developers are required to implement a certain pattern this will influence the final product cost.

5.2 Definitions of quality attributes

To have a clear frame of reference when assessing the influence of patterns on quality attributes of the architecture we must specify what our definitions of the desired quality attributes are.

These quality attributes often complement each other or an improvement in one attribute conflicts with another. We consider each attribute in isolation to avoid making trade-off decisions in the analysis phases. At implementation time it is important to determine the importance of each attribute for the implementation at hand.

An overview of the definitions of the discussed quality attributes is available in table 5.1.

5.2.1 Security

We defined security as the ability to isolate tenants from each other and the impact of security breaches in custom parts of the system on the other parts of the system. Due to the mult-tenant nature of Software as a Service the separation of data as well as isolation of (virtual) components is a key concern for security (Subashini & Kavitha, 2011). A pattern can be evaluated on its security impact by analyzing how the proposed implementation separates data from multiple tenants and what failure-scenarios exist for this approach. If for example a mistake in a described part of the application leads to a failure to access any data at all instead of access to data from different tenants this pattern could be considered more secure.

5.2.2 Performance

We base our definition of performance on the efficiency, the utilization of computing, storage and network resources by the application (International Organization for Standardisation, 1991), at a certain level of usage by clients.

Pattern implementations can have different requirements for extra resources. When analyzing performance we will not consider how the additional resources could be added to the system because that's part of the Scalability attribute.

5.2.3 Scalability

We define scalability as the relative increase in capacity acheived by the addition of computing, storage and network resources to the system (Nussbaum & Agarwal, 1991) as well as the flexibility with which these resources could be added to the system.

Scalability differs from performance by considering only the relative increase and not the absolute amount of resources. This relative increase may seem straightforward at first, doubling resources could double capacity. However in actual implementations increasing resources requires a system that is able to deal with using these extra resources effectively to achieve actual scalability.

Resources can be added to a system in two ways, either by increasing the available resources in a single system (e.g. doubling network connection speed or storage capacity) or by introducing extra systems (e.g. adding a second server). These two approaches are considered vertical scalability, adding more resources and thus vertically increasing the resources in a single server; and horizontal scalability, adding extra systems which must then share the load.

There is a natural limit to vertical scalability, faster components are not always available or the cost might be prohibitive because special components are only produced in much smaller quantities. This limit does not apply to horizontal scalability, making horizontal scalability a desirable property of systems that require high scalability.

Achieving perfect horizontal scalability has several challenges in coördination of nodes in a system. All nodes need to collaborate, communicate and possibly synchronize state. These extra overhead-costs could limit horizontal scalability.

5.2.4 Maintainability

We define maintainability as the ease with which the system can be extended and potential problems can be solved. The ease with which the system can be extended or modified is defined as changeability in ISO 9126 (International Organization for Standardisation, 1991), a sub-characteristc of maintainability. The ease with which potential problems can be solved is based on analyzability, which is also a sub-characteristic of maintainability in ISO 9126. Both can be influenced by the implementation of a variability pattern if for example the pattern requires modules to be more inter-connected than before.

Quality Attribute	Definition
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 acheived 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

Table 5.1: Desired quality attributes in business SaaS

Maintainability is also applicable to the variations or extensions created using the implemented pattern. When changes the application require re-implementation of all customizations this makes changes to teh application very costly. Such a system would score low on maintainability.

5.2.5 Implementation effort

We define implementation effort as the effort required to implement and deploy a specific system. This applies to both the initial implementation of the pattern as well as the implementation effort required for implementing a customization using the options provided by the pattern.

One pattern could affect only a small part of the system with relative ease of implementation, while another requires extensive change to multiple or even all parts of a system. The implementation effort for the former is low, while the implementation effort for the latter is high.

Excluded from the implementation effort attribute is the cost of later supporting or re-implementing the customizations when other parts of the application change. This is considered part of the maintainability of a system.

Chapter 6

Datamodel extension

6.1 Problem description

Organisations within the same market strive to differentiate themselves from their competitors, which often results in different working processes each with specific requirements for the supporting software systems. Across markets and jurisdictions there are differences 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 thus increases the cost per client. A software product that provides enough variability on the datamodel to meet organisation specific requirements will decrease cost and attrackt clients that cannot currently be serviced by software products unable to meet their specific requirements. Extension of the datamodel by creating additional fields to store data that is specific to an organisation or their working processes is a common requirement (Sun, Zhang, Guo, Sun, & Su, 2008).

An example of a requirement for an ERP application would be:

> ”Users in the bookkeeping-department need to store the tax-id
> of the local distributor of international suppliers”.

This requirement could be included in the development process of a tailored solution or it could be standard for an ERP application designed specifically for markets where it is common practice or even a requirement by law to record financial details of local distributors instead of the main supplier.

In case of standardized software where this requirement is not met by the default installation of the software an extension of the existing datamodel is required. Figure 6.1 depicts the envisioned functional situation, storing

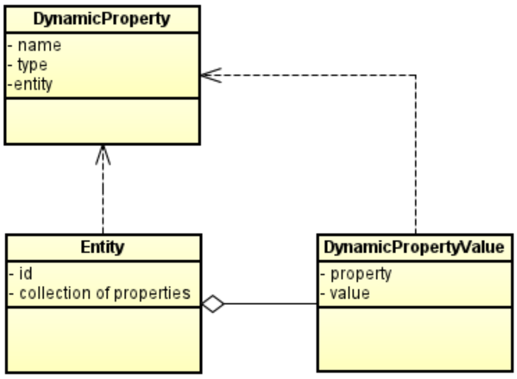

Figure 6.1: Functional model of datamodel extension

custom properties of entities in the domainmodel. The depicted Entity is the original entity in the application domainmodel which has a relation to a DynamicProperty. This property is configured for a specific tenant and holds settings like a name and expected data-type, in our example we could define the name to be ”Local tax ID”, type ”Number” or ”Text” depending on the type of tax-id specified in the requirements and entity ”Supplier” which references an entity in the original domainmodel of the application.

When using the application the user is presented with this extra option to store a Local tax ID, which is handled by the application backend by creating a DynamicPropertyValue which references the Local tax ID property and stores the value for a specific Entity (the Supplier in our example). Each Supplier object is now associated with a DynamicPropertyValue for the Local tax ID field.

6.2 Datasource Router Pattern

The first pattern applicable to datamodel extension is the Datasource Router pattern. This pattern was observed in a multi-tenant business reporting application which allows users to add extra fields to the records describing company specific information. Clients used these fields to track additional identifiers which were later used to link reported information to other systems. The platform used was based on Java-enterprise with a SQL-based datastore.

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

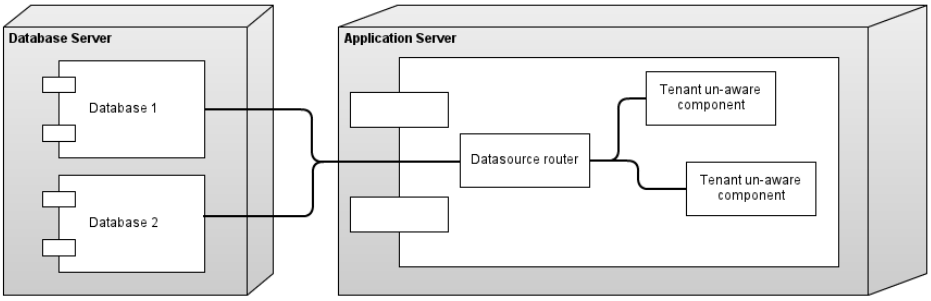

Figure 6.2: The System model for Datasource Router pattern

other components can thus work without being aware of the fact that the application is actually serving multiple tenants using different databases.

6.2.1 System Model

The system model describes the overview of the system when implementing the Datasource Router pattern, this model is included in figure 6.2. As shown the application uses multiple separate databases (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 (a query will always return results for one and only one tenant, because it is sent to a database which contains only data for a single tenant), the components thus have no need to be aware of multi-tenancy in querying the data.

6.2.2 ER-diagram

The database schema for the Datasource Router pattern is very simplicistic. On the schema level, no multi-tenancy is involved as each tenant uses a separate database or schema. The ER-diagram in figure 6.3 depicts the simplicity, the entity is self-contained, no extra entities for custom properties are used the only addition are the extra fields for custom-properties added directly to the table.

Depending on the features available in the implementation language the implementor could use different techniques for mapping these custom properties to the domain objects as used by the application. If the language supports extending objects at runtime then the custom properties could be added to the domain objects on a case-by-case basis (e.g. languages like Ruby have the ability to add fields to an object at runtime). If the implementation

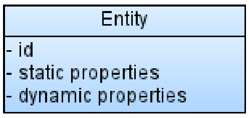

Figure 6.3: The ER-diagram for Datasource Router pattern

language does not have this capability or if the implementor decides not to use that capability the custom properties need to be explicitly included in the domainmodel.

One method of including the properties in the domainmodel explicitly is to add a map datastructure to the object that contains a mapping from property to value for each property of this entity. That datastructure could then be interated to display properties in the interface or searched for a specific property when the value is necessary somewhere.

6.2.3 Sequence-diagram

The interaction between tenant-unaware components and the database goes through the Datasource Router, the sequence diagram in figure 6.4 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

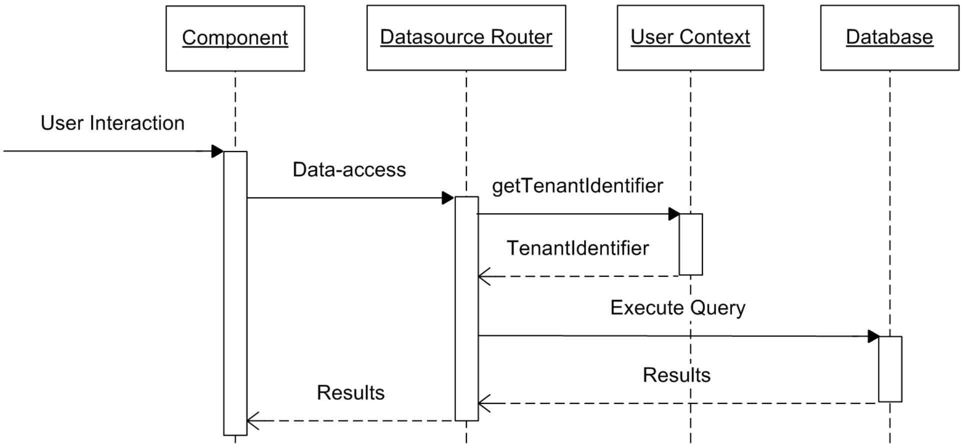

Figure 6.4: The sequence-diagram for Datasource Router pattern

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 doesn't 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.

6.3 Custom Property Objects Pattern

The second pattern for implementing datamodel extension is the Custom Property Objects pattern. This pattern was encountered in a SaaS based CRM application for small to medium sized enterprises. Clients were free to configure their own dynamic properties from a self-service interface. The fields allow for different datatypes like free text fields, limited drop-down selection boxes and boolean values. Depending on the datatype selected by the client the application uses appropriate interface components to allow entry and querying of the dynamic properties. The most common use-case in this application was the addition of extra fields for contact details such as the emailaddress of a personal assistant and the storage of workfield specific information for contacts such as the specific skillset for each contact in the instance of an employment agency.

When implementing this 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.

6.3.1 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. The system model is depicted in figure 6.5, 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 query for data only for the requested tenant or filter data while processing to get results only for the current tenant.

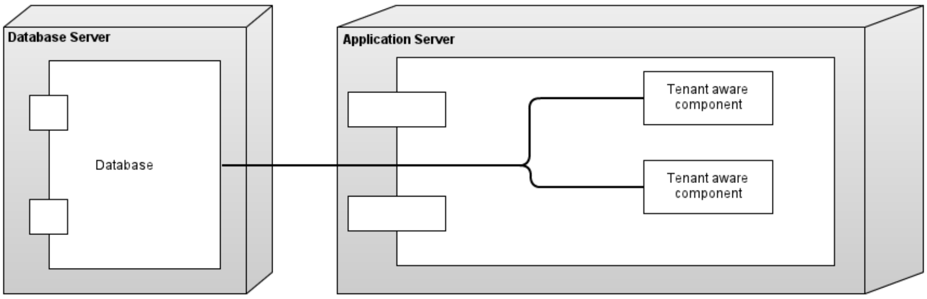

Figure 6.5: The System model for the Custom Property Objects pattern

6.3.2 ER-diagram

The added complexity of the Custom Property Objects pattern is in the database schema, extra properties are modeled at the meta-level to accomodate for any future properties while maintaining a static schema. This requires additional tables to define the properties and store their values in a generic way.

The pattern adds two tables to a common schema to hold definitions of the properties to be used and to store the values. Comparing the ER-diagram in figure 6.6 to the original functional model in figure 6.1 reveals that this pattern closely resembles the desired functional situation. The Entity has a relation to a DynamicProperty which defines the name and type of a custom property, the DynamicPropertyValue table then stores the values associated to those properties for each Entity.

The Property Objects that are created as a result of the implementation of this pattern can be used as-is in the application which makes the interaction between custom properties and other components in the application explicit. A component that needs access to custom properties for a specific Entity

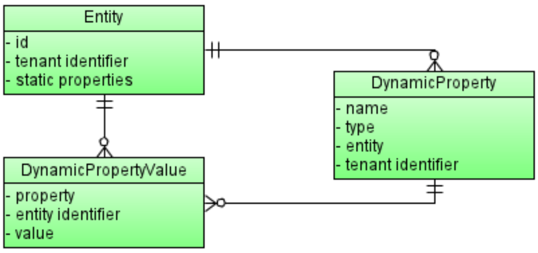

Figure 6.6: The ER-diagram for the Custom Property Objects pattern

has to use the DynamicProperty objects to determin which properties are available and then read the DynamicPropertyValue table to determine their values.

It is up to the implementor to determine whether the DynamicProperty will always be read when reading an Entity (effectively isolating the concern of retrieving the DynamicProperty to the data-access part of the application) or whether components can query a DynamicProperty when necessary and ignore the DynamicProperty when it is not necessary to handle the current request.

6.3.3 Sequence-diagram

As a result of using a single database for all tenants the other components in the application need to be aware of the context in which they are working. When retrieving data the components need to filter the results to only show data for the curent tenant.

The resulting interaction from component to database is depicted in figure 6.7. 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 boundariers.

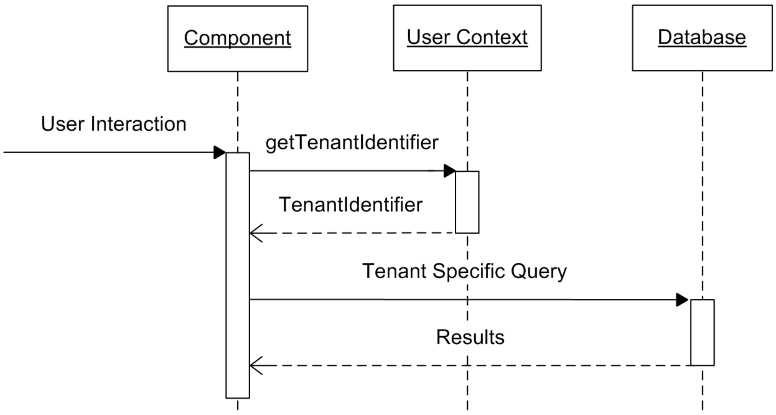

Figure 6.7: The sequence-diagram for the Custom Property Objects pattern

Chapter 7

Adding behaviour

7.1 Problem description

Software products not only need to offer a datamodel that fits an organisation's requirements, software behaviour also has to meet those requirements. When tailor made software is developed it is possible to set the requirements to exactly match the processes of a specific organisation. For a standard software product there can be steps or actions missing from the processes as modeled at design time which are required for the actual processes of potential clients using that software. Organisations will then require extra steps or actions to be taken at certain points in a process.

An example requirement for the ERP system of a manufacturing company requiring additional behaviour to be executed for specific company processes:

> "Send a notification to the department responsible for transportation if tomorrows batch will be larger than 30 boxes."

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 behaviour in certain parts of the application we need an application that allows configuration of this behaviour and the points at which it should happen. This functional situation is modeled in figure 7.1, the envisioned functional situation. The StandardComponent is a normal component of the software with default behaviour, this component has a set of ExtensionPoints. An ExtensionPoint is a location in the normal behaviour where there is a possibility to add or change behaviour. This behaviour is specified in an ExtensionComponent, which contains the actual behaviour that is to be executed at the specified ExtensionPoint.

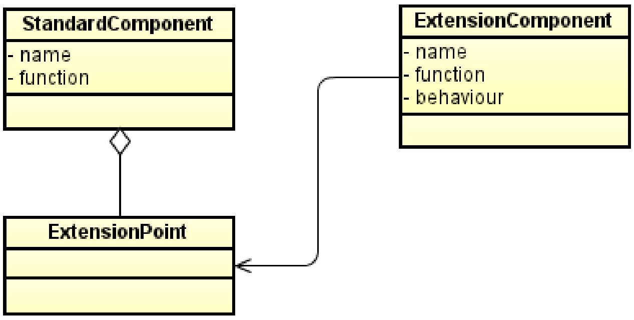

Figure 7.1: Functional model for adding behaviour

7.2 Component Interceptor Pattern

The Component Interceptor pattern was observed in a logistical planning system with extensive possibilities for adding and modifying behaviour for specific customers. This pattern prescribes the definition of extension points in the application as interceptors for requests to standard components. These interceptors then have the ability to decide what to do before and after passing on a request to the original component or even decide not to forward the request to the original component at all. Any additional behaviour is executed in these interceptors in-line with normal request processing either before or after the normal logic.

7.2.1 System Model

The system model for the Component Interceptor pattern as depicted in figure 7.2 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 standard component 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 would require 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 could involve a performance penalty (Carpenter, Fox, Ko, & Lim, 1999).

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

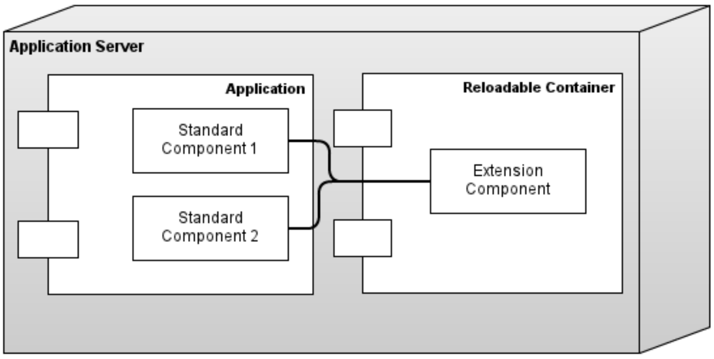

Figure 7.2: The system model for Component Interceptor pattern

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 sourcecode on the fly. Other platforms require special provisions for reloading code, such as OSGi for the Java platform or Managed Extensibility Framework (MEF) for the .NET platform.

7.2.2 Sequence-diagram

Figure 7.3 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, defeating the purpose of runtime variability. As depicted, multiple interceptors can be active for each 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 intercepetors have finished executing.

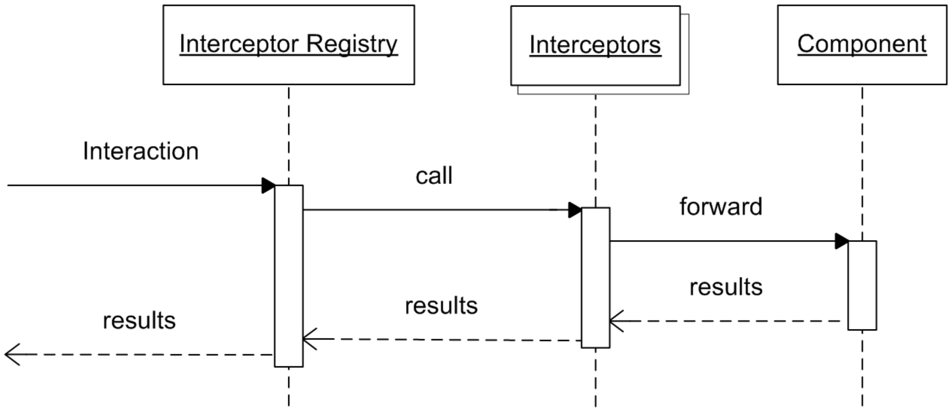

Figure 7.3: The sequence-diagram for Component Interceptor pattern

7.3 Event Distribution Pattern

In the event distribution pattern the application generates events at extension points which are distributed by a broker. We observed this pattern in a multi-tenant webbased application for the support of medical staff. Behaviour variability of the application was used to implement working processes of different healthcare institutions and different types health-services.

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.

7.3.1 System Model

The system model in figure 7.4 depicts the distributed nature of the 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

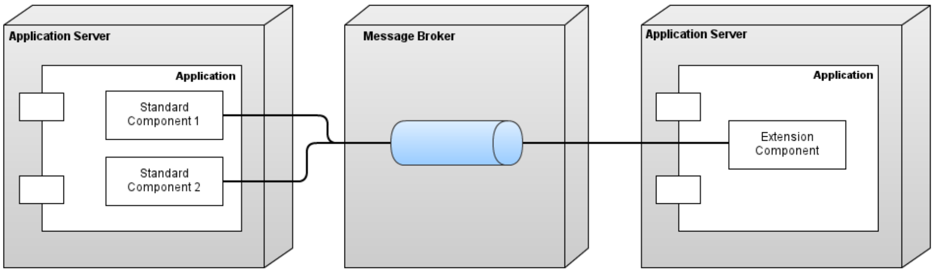

Figure 7.4: The system model for Event Monitoring Pattern

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.

7.3.2 Sequence-diagram

Standard components publish events to the broker as depicted in the sequence diagram in figure 7.5. 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 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 extension component to stop standard behaviour from happening. In the observed system this was solved by allowing extension components to execute a compensating action in their listener. The compensating action is send 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

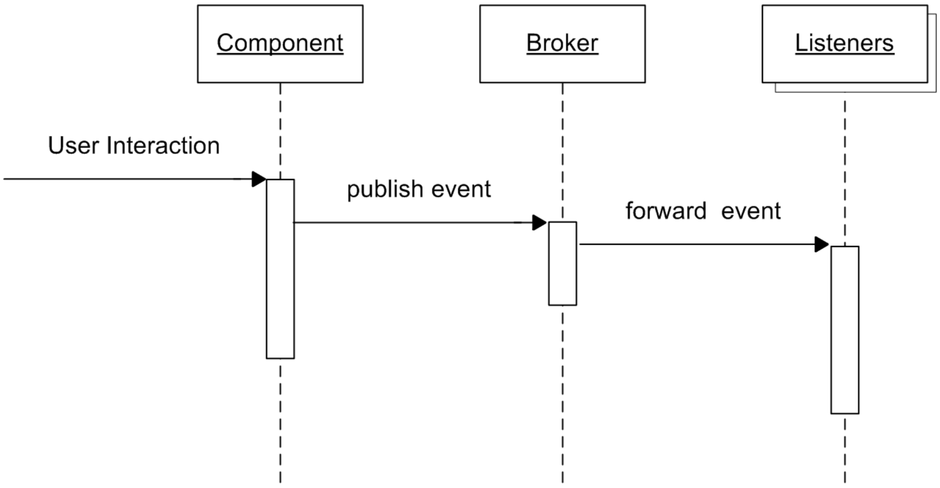

Figure 7.5: The sequence-diagram for Event Monitoring pattern

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 extension component.

Chapter 8

Analysis

This chapter describes the analysis conducted on the patterns which are described in chapters 6 and 7 using the criteria defined in section 5.2. The analysis is organized by subject area, starting with patterns for datamodel extension followed by the patterns for adding behaviour.

8.1 Patterns for datamodel extension

Chapter 6 describes two patterns for datamodel extension, the datasource router pattern in section 6.2 and the custom property objects pattern in section 6.3. In this section we analyze the properties of both patterns on security, performance, scalability, maintainability and implementation effort, the criteria as defined in chapter 5.

8.1.1 Security

Security is defined in section 5.2 as the ability to isolate tenants from each other and the possible impact of security breaches in custom parts of the system on the other (standard and custom) parts of the system. 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.

The custom property objects pattern on the other hand relies on the developers to write queries in all components to only return data from the right tenant. When no precautions are taken in the development and testing process the possibility of accidentally mixing data from multiple tenants is higher than in the datasource router pattern. A forgotten filter in this pattern will result in users receiving data from other tenants that should never be visible to them. When implementing this pattern it is thus 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.

Datasource router
✓ Natural separation of datasets ✓ Single point of selecting correct datasource ✗ More datasources to secure and maintain
Custom property objects
✓ Only a single datasource to secure and maintain ✗ Risk of losing dataseparation with programming errors

Table 8.1: Security properties of datamodel extension patterns

8.1.2 Performance

Performance is defined in section 5.2 as the utilization of computing, storage and network resources by the application at a certain level of usage by clients. Performance characteristics differ from scalability characteristics, discussed in the next section, by only concerning the use of resources and not the distribution of or dependencies between resources.

The custom property objects pattern uses only a single large database or schema which this allows the database server to allocate all resources like memory and caches to that single database or schema. The datasource router pattern requires a separate database or schema for each tenant which depending on the database system used could 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 very generic, storing all data in a generic field type 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.

Datasource router
✓ Correct data-types allow for optimizations ✗ Resource partitioning across separate schemas
Custom property objects
✓ Full resource utilization across all schemas ✗ Loss of optimizations due to lack of type information

Table 8.2: Performance properties of datamodel extension patterns

8.1.3 Scalability

As described in section 5.2 scalability is essential for the SaaS businessmodel. Scalability is defined as the flexibility to add computing, storage and network resources to the system and the relative increase in capacity achieved by the addition of these resources. For example adding a second database server to a system designed to only query a single server will result in no gain of capacity, such a system scores low on scalability.

Two directions 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 coördination of nodes in a system. In practice this coördination costs resources, which makes it hard to achieve lineair scalability in systems that require coördination 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 which 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 thus 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 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 becare stored on those new systems. There is virtually no overhead involved in this addition, because no extra coördination is required between the database systems servicing data for separate tenants.

For the property objects pattern this is not the case, it 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.

Datasource router
✓ Natural scalability due to separate schemas ✓ No need for scalability support in database
Custom property objects
✗ No inherent scalability in pattern structure ✗ Requires database system capable of scaling

Table 8.3: Scalability properties of datamodel extension patterns

8.1.4 Maintainability

In section 5.2 maintainability is described as the ease with which the system can be extended and potential problems can be solved. If for example all components are interconnected and changes in a single component might affect all other components in the system this is less maintainable than a system with clear separation of concerns in which modules can be changed without effects on other modules. The extra effort involved in making sure other modules are not negatively affected by the changes will increase the efforts involved in extending functionality or solving problems.

When extending the application with new functionality both patterns require that the new functionality is aware of any customized properties present on the objects it deals with. This is for example particularly important when dealing with copying objects or dealing with a versioning system. Both operations need to take all custom properties into account to avoid losing values.

For the datasource router pattern this involves creating technology that is able to determine all database-schema variations and correctly copy 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 normale 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 if this it is harder to isolate the root-cause of the problem, a necesary step in solving it. 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 possible schema-variations which must be explicitly handled correctly and tested.

8.1.5 Implementation effort

Defined in section 5.2 as the effort required to implement and deploy a specific system, implementation effort is a cost-driver for a SaaS application. There are two factors involved in implementation effort for datamodel extension patterns, the initial effort of implementing the pattern in the system and the

Datasource router
✗ Large number of possible database schemas must be tested ✗ Problemsolving requires schema variants to be included
Custom property objects
✓ Single static database schema ✓ Custom properties can be handled with generic shared code

Table 8.4: Maintainability properties of datamodel extension patterns

effort required for implementing variability requirements using the pattern.

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 datamodel. This is equal for both patterns and thus of no influence in a comparison on implementation effort.

Datasource router
✓ Central component to handle all data-access ✗ Custom properties must be handled in all components
Custom property objects
✗ Requires adaption of data-access in all components ✗ Custom properties must be handled in all components

Table 8.5: Implemenration effort properties of datamodel extension patterns

8.2 Patterns for adding behaviour

Chapter 7 describes two patterns for adding behaviour to an application to satisfy variability requirements. The first pattern presented is the component interceptor pattern described in section 7.2, followed by the event distribution pattern described in section 7.3. We will analyze both patterns on security, performance, scalability, maintainability and implementation effort, the criteria as defined in chapter 5.

8.2.1 Security

In section 5.2 security is defined as the ability to isolate tenants from each other and the possible impact of security breaches in custom parts of the system on the other (standard and custom) parts of the system. When adding new behaviour to the application there is always the possiblity of introducing new security vulnerabilities. This is an inherent risk of extending an application. The variability patterns do however influence how much larger the attach-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 behaviour 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 7.2 in section 7.2.1. 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 as described in section 7.2.1.

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 7.4 in section 7.3.1. This execution in a separate

Component interceptor pattern
✗ Extension components execute within application scope
Event distribution pattern
✓ Isolation of extension components ✓ Full traceability of actions by extension components

Table 8.6: Security properties of patterns for adding behaviour

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 behaviour is fully traceable including the original values (Fowler, 2002).

8.2.2 Performance

Performance is defined as the utilization of computing, storage and network resources by the application at a certain level of usage by clients in section 5.2. Variability patterns can have a different impact on the necessary resources to support the same level of clients. As put forward in section 8.1 performance differs from scalability in that it only concerns the amount of resources consumed and not the characteristics of how these resources must be added to the system.

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 which 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 meganisms 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,

Component interceptor pattern
✓ Direct execution of extension components
Event distribution pattern
✗ Network overhead for calling extension components ✗ Broker system requires extra resources

Table 8.7: Performance properties of patterns for adding behaviour

these steps add non-trivial cost to the operations (Thekkath & Levy, 1993).

8.2.3 Scalability

In section 5.2 scalability is defined as the flexibility to add resources to a system and the relative increase in capacity achieved by addition of those resources. Section 8.1.3 provides a more in-depth description of the differences between horizontal and vertical 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 eventhandlers 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.

Component interceptor pattern
✗ No independent scaling of extension components ✗ Does not scale to high number of extension components
Event distribution pattern
✓ Independent scaling of extension components ✓ Extension components cannot delay standard components ✗ Requires scalable message-broker system

Table 8.8: Scalability properties of patterns for adding behaviour

8.2.4 Maintainability

Section 5.2 defines maintainability as the ease with which the system can be extended and potential problems can be solved. If for example modules are clearly separated and independent it will be easier to extend functionality in one of the modules without side-effects on other modules.

When adding behaviour the an application, maintainability is also affected by the necessity to make sure future extensions and modifications are compatible with any custom behaviour implemented for tenants. This is a trade-off between the flexiblity and depth with which ExtensionComponents can affect the application and the impact that changes to the appliation 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 and 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 fro 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, & Draheim (2007).

Component interceptor pattern
✗ Tight coupling of extension components
Event distribution pattern
✓ Loose coupling of extension components

Table 8.9: Maintainability properties of patterns for adding behaviour

8.2.5 Implementation effort

Implementation effort is defined in section 5.2 as the effort required to implement and deploy a specific system. When implementing a pattern for adding behaviour 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 (see figure 7.3).

In the event distribution pattern a message broker system must be setup to handle the event-messages flowing from StandardComponents to ExtensionComponents. As described in section 8.2.3, these systems are available off the shelf. 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 behaviour when certain conditions are met.

Component interceptor pattern
✓ Direct communication with standard components ✓ Access to all data by design
Event distribution pattern
✗ Requires the setup of a message broker system ✗ Requires a separate mechanism to communicate with the application

Table 8.10: Implementation effort properties of patterns for adding behaviour

Chapter 9

Evaluation

9.1 Expert selection

We asked experts from the field of software architecture to review the patterns, their practical applicability and our analysis done in chapter 8. The experts were selected from the network of the authors based on experience in real-world software architectures for multi-tenant SaaS applications.

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 inclues a multi-tenant administrative application storing security sensitive data for multiple organizations. Security and isolation of tenants is an important area of expertise necessary for this project.

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 coördinated processing of large volumes of data to calculate metrics used for marketing. Performance and scalability are important areas of expertise for their product.

9.2 Patterns for datamodel extension

The experts were asked to review both problem domains independently, first datamodel extension, followed by adding behaviour, then provide their comments and experiences in these fields. The first problem domain are patterns for datamodel extension. Both experts have experience implementing datamodel extension in real-world projects. The first expert has also seen similar approaches used in several non-multi-tenant projects. His view is that these concerns are also applicable to a multi-tenant context because multi-tenancy adds another layer of complexity which doesn't shield from the general problems of implementing an extensible datamodel. Most of the issues and difficulties from an extensible datamodel are multiplied in a multi-tenant context making the issues even more important. How both patterns affect the impact and possible solutions to these challenges is part of the discussion per pattern.we concluded that

9.2.1 Datasource Router Pattern

The datasource router pattern, described in section 6.2, proposes a setup where separate datasources are used for all tenants using a central datasource router to determine which datasource to use for each operation. This pattern was analyzed in section 8.1 discussing the influence on security, performance, scalability, maintainability and implementation effort. We asked the experts to provide their opinion on these same characteristics without reading our analysis. We then discussed our analysis and the similarities and differences with their views.

Security

We concluded that the datasource router pattern has a built-in separation of data from multiple tenants. Both experts agreed on this, stating that the separate schemas provide a level of separation that naturally makes it easier to keep data from leaking. The first expert however warned that adequate validation of the separation is still necessary, especially in batch-operations and background jobs which often don't run in a clear single-tenant context. Batch jobs accessing multiple tenants should be implementated carefully to make sure they always keep the right context-data for tenants to access the right datasource.

The second expert shared a similar concern on carefully controlling the datasource router, especially when automatically binding the current context to thread execution as would be an obvious implementation in webapplications. He explained a situation in which a webapplication associates an execution-thread to a single datasource (tenant-context) after which that thread is returned to a shared thread-pool without resetting the association. This could for example happen when the thread encounters an exception and normal cleanup code is not executed. If this thread is later re-used for another request the original association could (depending on implementation details) still be active which would chause a dataleak between contexts.

The problems mentioned by both experts have a common source, associating code execution with the right context should be carefully checked and setup in a fail-safe manner in which the application stops execution if for example a context is already selected while the code tries to select another context. The first expert mentioned Aspect Oriented Programming (Kiczales, Lamping, Mendhekar, Maeda, Lopes, Loingtier, & Irwin, 1997) as a possible technology to automate these find of checks.

On the system level both experts disagreed with our assessment that adding more database-systems would add a larger attack-surface to the application. This maintenance and configuration should be automated in any large-scale application leading to a single homogeneous setup without additional attack-vectors.

Performance

In section 8.1 we discussed the performance of both datamodel extension patterns. We concluded that the datasource router pattern requires a database engine to split resources between several databases which could be detrimental to performance. The first expert agreed on this observation and provided insights in another area where this impacts performance. Not only is the database engine forced to split resources between caches, in most cases the application is also forced to use more connections to the database engine to acess all separate schemas. The second expert did not consider this an important performance limitation because database engines like MySQL and its derivatives handle multiple schemas in a single engine without the need for separate connection-pooling. We should therefore consider this an implementation dependent factor.

The first expert considered the split schemas created by the datasource router pattern an opportunity to optimize the database engines for the specific query pattern of each tenant. We defined variability only as the extension of the logical datamodel itself, however as pointed out it is possible to use the same techniques for variability in the physical datamodel allowing for optimizations on a per-tenant basis matching the specific query pattern of each tenant. An example of this performance optimization is the addition of extra indexes for data that is frequently queried by a specific tenant while not adding the overhead of handling those indexes for tenants that will not query that specific data.

On performance the second expert agreed with our analysis that resources would be split between databases which avoids using all resources optimally. He proposed a solution in which the multiple-schemas are all held in the same database-engine resulting in optimal use of resources across all schemas while still keeping the separation that is intended by the pattern.

Scalability

Our analysis of scalability in section 8.1 includes a description of the differences between horizontal and vertical scalability. Both experts agree on the definitions and differences discussing horizontal scalability separately from vertical scalability. Both agreed that vertical scalability is not impacted in a significant way by the datasource router pattern which corresponds to our conclusion.

The first expert agreed on our analysis that horizontal scalability is very natural to the datasource router pattern because of the inherit split in data. He did however comment that this only covers scalability to a higher number of tenants, a single database powerful enough to handle all users of the largest tenant is still required. The second expert mentioned this as a downside of the datasource router pattern as well because the impression that scaling is not a problem could arise from the ease with which this pattern scales to higher number of tenants.

The second expert also shared examples of issues that arise when the datasource router pattern is used for very large volumes of data. To keep the system working optimally is will sometimes be requires to rebalance the databases across the available servers. This could be a trivial task for small datasets, however the time to move a very large dataset from one server to another is non-trivial. If no built-in support for this is included in the database engine it could result in the application being unavailable to a tenant for the duration of the moving process. For certain applications and datasets this could be unacceptable.

On the subject of scalability the first expert also warned implementors to clearly distinguish scalability from scaling. Initially providing scalability in the application architecture is important, however scaling up the application without an actual need for this extra capacity is detrimental to development time and resources. His example is the ability to first run all separate schemas on a single system, having scalability in the architecture without actually scaling to a second and further database servers.

Maintainability

The experts involved in our review both have experience in maintenance of long-lived application architectures. Both agreed on our position in section 8.1 that allowing broad modification of the datamodel in the datasource router pattern could hamper maintainability of the application considerably. The second expert pointed out that limiting the allowed modifications might seem counterproductive to our objective of implementing variability requirements however a strict set of allowed variations is similar in results to what the other pattern discussed, custom property objects (section 6.3), will allow.

The first expert recommended the implementation of code dealing with special properties via aspect oriented programming because dealing with extra properties is a cross-cutting concern. This could decrease the effects of this pattern on maintainability because the modifications only need to be handled in a single area. The second expert discussed a similar situation in

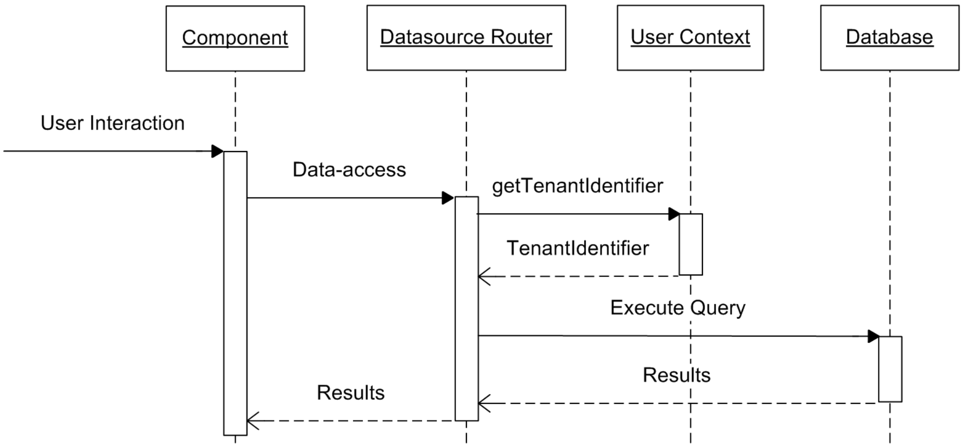

Figure 9.1: The sequence-diagram for Datasource Router pattern

which a generic approach was used to handle the maintainability difficulties in dealing with custom properties. He did however agree on our analysis that this has an impact on maintainability and problem solving, because if the handling of custom properties is abstracted away into a generic component it is harder to understand what code does without looking up the functionality of this generic component as well.

Implementation effort

We defined implementation effort in section 5.2 to encompess two components, the initial effort required to implement the pattern as well as the effort required to implement the actual variability when using the pattern. In the analysis we found that the datasource router pattern requires the implementation of a datasource router component and facilities to automatically manage the required database instances. The other components of the application need not to be aware of how multi-tenancy is handled.

The first expert agreed on the concept that in a well designed architecture the other components do not need to be adapted to be aware of the use of multiple schemas in the datalayer. Concepts from the datalayer should not leak into other components of the application which abstrcts the schema routing away from other components. He did however warn that the pattern could introduce a new concern for certain parts of the application, because code must provide a user context for the datasource router to be able to select the correct datasource.

The user context is apparent in figure 9.1 (equal to figure 6.4 in the pattern description). The datasource router determines the tenantIdentifier through a call to the user context. This user context could be implemented in several ways. For web applications it would be natural to bind this context to the thread executing the current request. The expert warned that in more complex applications there is a significant implementation effort in making sure components like background-processes execute in the correct context as well.

The second expert did not initially consider this a problem. He described an application configuration in which the tenant-selection is bound to the security context of a component. If background-processes are designed to run within the security-framework of the application (e.g. with appropriate rolas and permissions) instead of being excempt from security constrains as being a system process then no extra effort is needed to implement a user context for these processes.

Considering the different positions from the experts it is likely that the implementation effort of keeping the user context requires attention when selecting this pattern. Depending on the implementation situation at hand and the current architecture of the application it might be considered a major part of the implementation effort or a small change to existing security-frameworks.

9.2.2 Custom Property Objects Pattern

The custom property objects pattern, described in section 6.3, describes a system in which a single datasources is used for all tenants and the extensions to the datamodel are stored as separate objects within this static schema. We analyze the security, performance, scalability, maintainability and implementation effort influence of this pattern in section 8.1. We asked the experts to provide their opinion on these same characteristics without reading our analysis. We then discussed our analysis and the similarities and differences with their views.

Security

In section 5.2 we defined security as the ability to separate tenants from each other and the impact of security breaches in custom parts of the system on the other parts of the system. Section 8.1 concluded that the custom property objects pattern has less natural separation of data than the datasource router pattern, both experts agreed in this. We described the critical nature of implementing a test and quality assurance system to ensure all data access is properly filtered. The first expert proposed Aspect Oriented Programming (Kiczales et al., 1997) as a possible means to implement automatic filtering of data. Filtering is not a substitute for correct querying, however a bug in a query will not cause a security breach if filtering is applied to all query results.

The second expert agreed with our analysis and considers this pattern less secure by default than the datasource router pattern. It is important in implementation cases to make sure that multiple layers of filtering and querying are implemented which catch programming errors and if possible generate alerts that can be used in quality control to fix errors.

Performance

Performance is defined as the utilization of computing, storage and network resources by the application at a certain level of usage by clients in section 5.2. We discussed the properties of both datamodel extension patterns in section 8.1, we found that the custom property objects pattern uses all resources for each tenant, leading to less partitioning of resources compared to the datasource router pattern. The first expert agreed on this, explicitly mentioning the partitioning of resources as an issue for the datasource router pattern. He did however also mention that for some applications the usage pattern might be different per tenant. Having all data in a single schema makes it harder for the database engine to detect common access patterns for optimization.

In the analysis section we found that the efficiency of queries on custom properties is dependent upon the design of the database schema. The sescond expert considered query efficiency in this type of schema to be a compromise of generic design to handle all types of properties which leads to lower query performance. Partitioning data within the database by tenant could increase performance because only a subset of the data is involved in each query.

The first expert weiged in on this property by describing the risks of doing join-operations on custom properties. The example provided is of an application where multiple object types used custom properties which could also be used to describe relations between those objects. This follows from the schema depicted in figure 9.2 (a copy of figure 6.6, used in the pattern description), all property-values are stored in the DynamicPropertyValue table which leads to recursive joins where a field is joined with itself. The expert described this as an undesirable situation which is underlined by Oracle in online advice on query performance (Oracle.com, 2008).

The second expert shared a similar concern stating that the value column of the DynamicPropertyValue table must be of a very generic type to be able to handle all types of custom properties that could be stored by the application. This field would use a variable length which has lower query performance than known length fields or use a very large field to store the largest possible value necessary for all custom properties, wasting space for smaller properties. Optimizations in the database engine for handling different data-types cannot be used for this field, because the database engine is unaware of the datatype actually stored in the field.

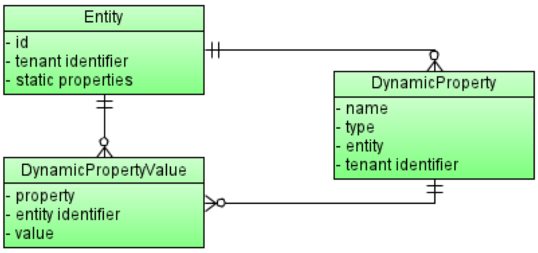

Figure 9.2: The ER-diagram for the Custom Property Objects pattern

Scalability

As described in section 8.1.3 two directions of scalability exist, horizontal and vertical. In analogy of the datasource router pattern both experts agreed that the custom property objects pattern has no significant impact on vertical scalability. The analysis focussed on horizontal scalability as the desirable property of Software as a Service as described in section 5.2. Our analysis concluded that scalability for this pattern is dependent upon the scalability options provided by the database system. The experts agreed, the first expert provided MySQL Cluster as an example of a scalable database engine that could handle horizontal scaling of the database engine across multiple nodes.

Our analysis described the requirement for scaling within the database engine as a limitation to the custom property objects pattern. The experts considered scaling the database engine external to the application a good solution. Moving the scalability burden from the application to the database engine specifically designed to handle scaling ensures that a technically sound strategy to data-layer scalability is used.

Maintainability

The custom property objects pattern has a smaller influence on maintainability compared to the datasource router pattern according to our analysis in section 8.1. The second expert agreed on this analysis, bringing up an issue in testing future versions of the software which require database modifications. This affects the datasource router pattern, because the changes could conflict with changes that have been made to the database schema for a single tenant. The custom property objects pattern does not have this problem because only a single schema is used and custom properties are explicitly defined in this schema.

The first expert partially agreed with this with regards to the model stored in the database, all properties are explicitl and no conflicts are expected. He did however share concerns with regards to the handling of custom properties in the user interface layer of the application. It is important for all components to be aware of the existence of custom properties and how they should be displayed or queried. This matches our analysis that in both patterns the rest of the application must be aware of the existence of custom properties. In the analysis section we described this as a common concern for both patterns, which is easier to handle in the custom property objects pattern. The expert disagreed on this, describing the possibilities of enumerating the database schema in case of the datasource router pattern. The second expert agreed on our analysis sharing an experience in which the datasource router pattern was used and database enumeration proved difficult because of the large amount of variations possible in a database schema.

On the subject of maintainability of queries and other components the second expert also shared the conclusions of the analysis section. During maintenance and extension of the application all queries must always be written in a way that is compatible with the shared database schema, continuously filtering all returned data to only use the data from the correct tenant. The first expert agreed on this maintainability burden, however he considered this a cross cutting concern, suitable for a generic solution aspect oriented programming. This shifts the handling of filtering and checking to the implementation stage of the pattern in which a generic component must be developed to handle all filtering.

Implementation effort

During our analysis of the patterns for datamodel extension on implementation effort in section 8.1 we concluded that the custom properties object requires the adaption of all queries to tenant specific queries. However the handling of custom properties themselves could be done using a generic handler querying the shared tables DynamicProperty and DynamicProperty-Value as depicted in figure 9.2. Both experts described implementation of the custom property objects pattern in an application that already used a single database schema for all tenant-data. In this scenario the concern of adapting all queries to use the correct filters for each tenant is not applicable to the pattern implementation.

We did not consider the implementation effort of the user context in our analysis. The user context is depicted in figure 9.3 (a copy of figure 9.3, used to describe the pattern in section 6.3). If we compare this to figure 9.1 the same user context implementation is necessary as is the case in the datasource router pattern. This leads to the same implementation effort concerns with regards to the setting of the correct context and adaption of any background or system processes to use the correct context as described in section 9.2.1 for the datasource router pattern.

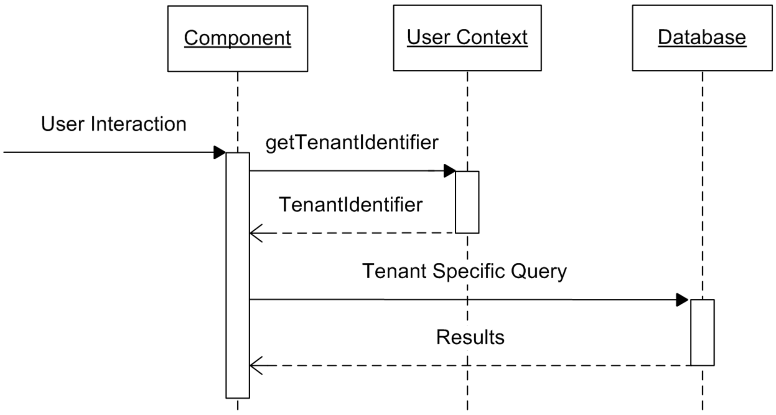

Figure 9.3: The sequence-diagram for the Custom Property Objects pattern

9.3 Patterns for adding behaviour

Subsequent to the review of the patterns for datamodel extension we asked the experts to review the patterns for adding behaviour. The experts were asked to review both patterns independent of each other as well as provide comments on the differences between the two. The differences are included in the review of the second pattern, the event distribution pattern, to provide insights in the tradeoff decisions needed while implementing the patterns.

Both experts have experience implementing variability to add behaviour in real world applications. The first expert has worked on application frameworks used in numerous applications supporting both the component interceptor pattern as well as the event distribution pattern. The second expert has consideral experience using the event distribution pattern in a large scale distributed application running across multiple geographical regions. He is familiair with the component interceptor pattern and has used it on a smaller scale.

9.3.1 Component Interceptor Pattern

The component interceptor pattern, described in section 7.2, prescribes a setup in which calls to extensible components go through an interceptor. As depicted in figure 9.4 (a copy of figure 7.3 in section 7.2.1) the interseptors are held in the interceptor registry. Each interceptor has the ability to change incoming calls, stop them, modify their parameters or forward them unaltered. The return value of the component also passes through the interceptors which provides the ability to modify or filte the values or add extra values.

We analyze the security, performance, scalability, maintainability and implementation effort influence of this pattern in section 8.2. The experts

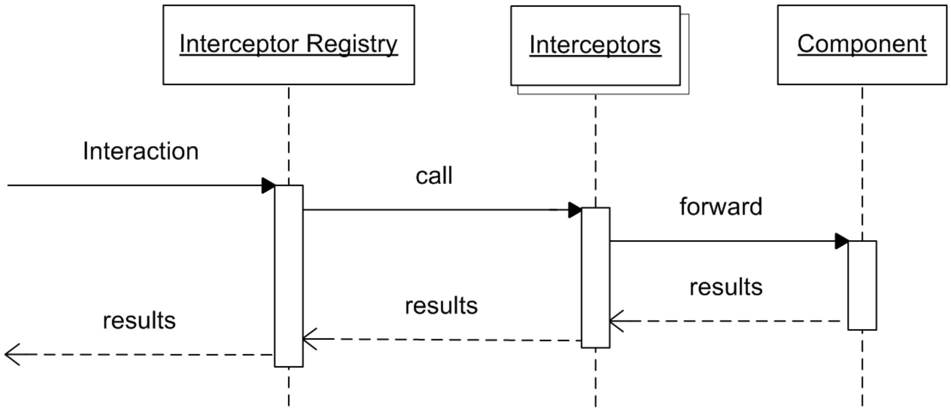

Figure 9.4: The sequence-diagram for Component Interceptor pattern

were asked to provide their opinion on these same characteristics without reading our analysis. We then discussed our analysis and the similarities and differences with their views.

Security

The security of the component interceptor pattern was analyzed in section 8.2. We concluded that the component interceptor pattern requires the code of each interceptor to run within the context of the application with full access to all parameters and values passed to each method call. The second expert agreed on this, the first expert noted that this would be the most common implementation however with more effort it would be possible to execute the interseptors in a separate context. He did agree that this would be an exceptional implementation with negative effects on the performance of this pattern.

The analysis also describes a concern in which the changes made by the interceptors are not explicitly visible in the data. This decreases traceability in case of security concerns, because it is unknown whether a user entered a specific value or whether it was changed by one of the interceptors. The second expert described an example case for the use of audit-logging to make changes by interceptors explicit. This could provide the necessary traceability as a separate implementation.

Performance

We defined performance as the utilization of computing, storage and network resources by the application at a certain level of usage by clients (section 5.2). The first expert noted that measured performance is not always equal to perceived performance while perceived performance is possibly

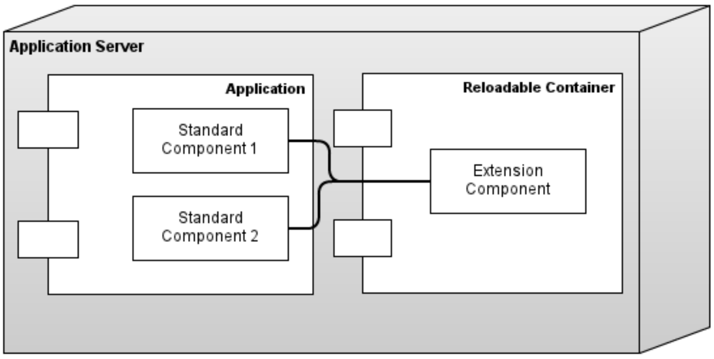

Figure 9.5: The system model for Component Interceptor pattern

a more important factor for software products. He described a case in which the perceived performance of an application by its users was improved considerably by executing certain tasks in the background while reporting back to the user immediately. The measured performance was equal or lower, because the tasks had to be scheduled and executed, however the perceived performance by the users was much better.

Our analysis in section 8.2 focussed on measured performance over perceived performance. We described the low overhead of executing interceptors within the context of the application, avoiding marshalling and unmarshalling cost for transferring the calls over network. Both experts agree on the low impact on measured performance for the interceptor pattern, noting that the interceptors must be developed to execute quickly because they will add to the response time for the user.

Considering the perceived performance the component interceptor pattern has a larger influence than implementations in which the additional behaviour is executed asynchronously without waiting time for the user. This is an implementation consideration depending on the type of behaviour that is added to the application and the sensitivity to latency for the user.

Scalability

The analysis in section 8.2 describes how interceptors are part of the application which cannot be scaled independently. This is depicted in figure 9.5 (equal to figure 7.2 included in the pattern description), a single application server executes both the standard components as well as the extension components.

Both experts agreed this poses a limitation on scaling the interceptor components. The first expert noted that the interceptor registry could be considered a routing component which determines which chain of interceptors is applicable to a certain reques. This requires a more complex interceptor registry, however it would increase scalability to a high number of interceptors because not all interceptors have to be processed for each request.

The second expert put emphasis on the scalability of the application as a whole. If the platform on which the application runs is already configured for scaling by running multiple instances of the application then the interceptors would piggyback on these existing meganisms. This does however not cover the problem indicated by the first expert where a very large number of interceptors could slow down the application with no way to scale out.

Maintainability

As defined in section 5.2 maintainability consists of two components, the ease with wich the system could be extended and with which problems could be solved. Maintainability affects both the application itself (the standard components) as well as the extension components created to implement variants of the software. Our analysis in section 8.2 describes how the component interceptor pattern influences maintainability by requiring the interceptors to be adapted when intercepted calls change. The first expert considered this only a small impact on maintainability for an application with well-designed service interfaces. The second expert agreed on our analysis, large numbers of variations need to be tested and adapted if the signature of a method call changes. Having well-designed service interfaces decreases the changes of this happening, but does not eliminate it completely.

The second expert also voiced a concern with finding the source of a problem which could be in interceptor code which is not available in the development environment. Based on this he recommended having an extensive testing and quality assurance setup that includes all interceptors from production systems. The requirement for testing interceptors for each change to the application decreases maintainability. The expert described an example case in which extension components were not tested together with a new release were subtle changes to the underlying API affected the results of a method call. In this case specifically a value for a missing proprety was changed from empty to a null-value, which caused some variants to fail silently. This should have been designed differently, however for practicality reasons not all interceptors were included in the development environment. The problem only became apparent in the last stage of quality assurance, causing a missed release deadline.

Implementation effort

We defined implementation effort in section 5.2 to encompess two components, the initial effort required to implement the pattern as well as the effort required to implement the actual variants when using the pattern. In the

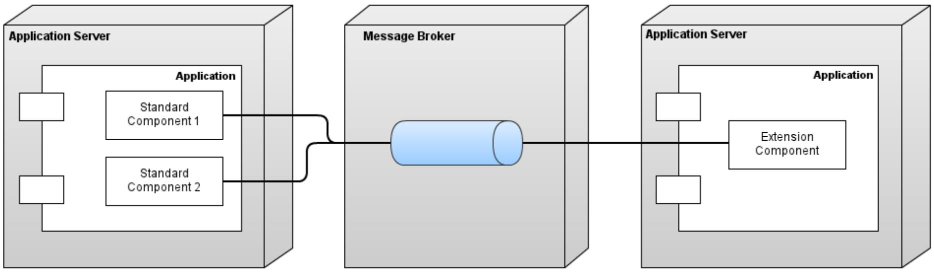

Figure 9.6: The system model for Event Monitoring Pattern

analysis phase we found that both patterns require efforts to determine which extension points in the application must be created. Both experts agree this is the most significant implementation effort for all patterns providing the ability to add behaviour. This is mostly determined by the amount of variability required in the application.

9.3.2 Event Distribution Pattern

The event distribution pattern, described in section 7.3 prescribes an architecture in which each extension point sends events that can be listened for by extension components to act on. As depicted in figure 9.6 (also used in the pattern description) the events originating in the standard components are transported by a broker to the extension components. The extension components reac to these events by executing additional behaviour. As described in the pattern description the extension components cannot change the event, each event indicates something that has already happened and cannot be changed. Extension components interested in changing the results of a standard component must do so separately by sending change commands.

Our analysis of the influence of this pattern on security, performance, scalability, maintainability and implementation effort is described in section 8.2. The experts were asked to provide their opinion on these same properties without reading our analysis. We then discussed the similarities and differences between our analysis and their views.

Security

The analysis section describes the ability to separate event handlers in the event distribution pattern from the standard components of the application as a possibility to increase security by providing better isolation. Both experts agreed on this conclusion, the second expert explained the possibility of executing the event handlers completely outside of the context of the application by providing a public API that delivers the events to a third party. This approach is taken by several commercial SaaS applications like Atlassian Jira, 37 Signals Highrise, Zendesk and others through services provided by Zapier.com. In this case the application implements the event distribution pattern in a way that delivers events through a webservice and clients can implement their own variability using external applications.

Compared to the component interceptor pattern the isolation of extension components outside the application context also allows for better traceability of the actions of a component. The first expert agreed on the possibilities of using event sourcing (Fowler, 2002) described in the analysis section. He explained the possibilities of using this to provide full traceability of actions of extension components.

Performance

In section 8.2 we discussed the performance of both patterns for adding behaviour. We defined performance as the utilization of computing, storage and network resources by the application at a certain level of usage by clients. Adding behaviour to an application naturally requires resources to execute this new behaviour. There are however differenes in the overhead of implementing the variability. As described in our analysis the event distribution pattern requires events to be transformed to a format suitable for transmision to a central broker. Both experts mentioned the extra resources needed to translate the data into a format suitable for transmission compared to the component interceptor pattern. The second expert explained research in the area of data-serialization to determine which data formats have the lowest cost of transferring over network. He warned for the costs associated with standard serialization formats such as the one built into the Java platform. Those methods require significantly more CPU resources while creating a larger datasize compared to specialized methods of serialization like Google Protocol Buffers (Google, 2008), MessagePack (Ohta & Colebourne, 2011) or Kryo (Grotzke, Sweet, & Levenstein, 2009).

The analysis also describes the extra resources necessary to operate a message broker system which has to handle all events generated by the operation of the application. The experts did not consider these resources in their initial review, however both agree this is part of the performance characteristics of the event distribution pattern.

Scalability

Our analysis of scalability in section 8.2 focussed on horizontal scalability, a more in depth description of the differences between horizontal and vertical scalability is included in section 8.1.3. The analysis focussed on horizontal scalability of both patterns for adding behaviour because vertical scalability is not impacted by the implementation of these patterns. Both experts agreed on this choice, reviewing the patterns from the perspective of horizontal scalability.

The scalability of the event distribution pattern, in contract to the component interceptor pattern, is independent of the application scalability. Both experts mentioned the possibilities of scaling the extension components separately from the application. Compared to the component interceptor pattern this allowes for scalability also in the number of extension components without affecting scalability of the application. The second expert also described the ability to use messaging or webservice APIs for controlling the system from the extension components as an opportunity for increased scalability. Because the event listeners execute in a different context all commands flowing back to the application could be routed through loadbalancing components.

The analysis describes the possibility of using open source of commercial message broker products like Fuse Message Broker, JBoss Messaging, RabbitMQ, Microsoft BizTalk, Oracle Message Broker or Cloverleaf. The experts are familiair with several of these products, the first experts recommended the addition of Zero MQ to the list of possible message systems describing a system in which this component was used to efficiently handle large streams of events.

Maintainability

We defined maintainability in section 5.2 as the ease with wich the system could be extended and with which problems could be solved. This affects both the application itself as well as the variants created via extension components. We analyzed the maintainability of the event distribution pattern in section 8.2. The analysis describes how the events in the event distribution pattern are decoupled from the standard components allowing for changes to the standard components without affecting the extension components. Both experts discussed the ability to change standard components without changing the events that are sent to ease maintainability compared to the component interceptor pattern.

The second expert described the use of extensible dataformats like XML or the aforementioned Google Protocol Buffers (Google, 2008) which makes it possible to extend events with new datafields without impacting existing extension components. The new components could still use the old dataformat without being affected by the new data, only upgrading to the new format when the newly provided data is of interest to them.

Both experts described the possibility to use a webservice based API to communicate back from the extension components to the application. In the analysis we proposed using existing webservice versioning techniques to allow for upgrading of the internal services without affecting the extension components. The second expert shared doubts about the practical feasibility

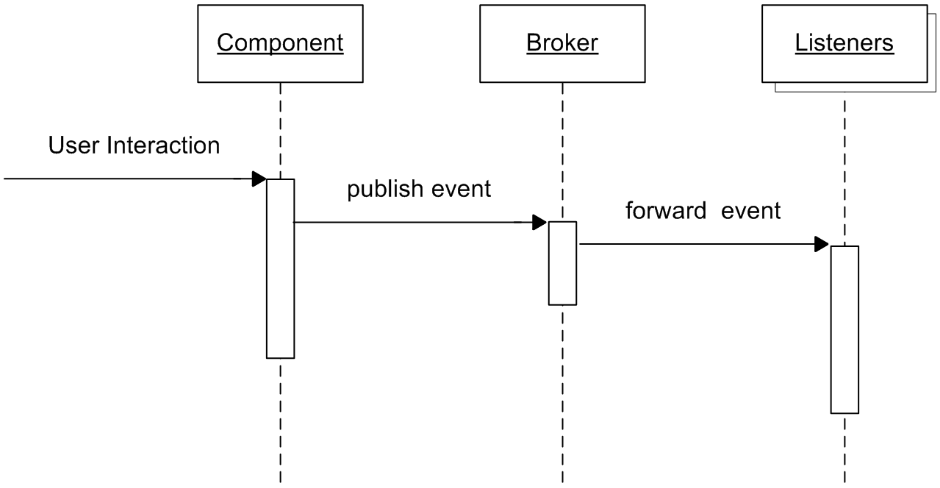

Figure 9.7: The sequence-diagram for Event Monitoring pattern

of versioned webservices, considering this is not practical for some projects. We should thus consider this a partial solution depending on the context in which the pattern is implemented.

The second part of maintainability, the ease with which problems can be solved, was not considered in depth in the analysis section. The first expert did not provide comments on this factor. The second expert described the decoupling between the event handlers and the standard components as beneficial to maintainability. In the component interceptor pattern a failure could be caused by an interceptor, requiring problem solving to include all configured interceptors. When using the event distribution pattern a failed in an extension component is independent of the application. Application failures cannot be caused by the event handlers, because (as depicted in figure 9.7) the event handlers are not in the normal call-flow of the application.

Implementation effort

We defined implementation effort in section 5.2 to encompess two components, the initial effort required to implement the pattern as well as the effort required to implement the actual variability when using the pattern. During the analysis of the patterns for adding behaviour in section 8.2 we determined that adding extension points to all standard components is a large implementation effort for both patterns. This is dependent upon the amount of variation required. The second expert agreed that implementing the extension points is the largest part of the implementation effort for both the component interceptor pattern as well as the event distribution pattern.

The first expert however provided an example of an architecture based on CQRS (see Young (2013) for a detailed description) and Domain Driven Design (Evans, 2003). In this design the application is based upon receiving commands from the user in a separate flow from querying data. Each change to the data is always handled through sending a command. In shuch a case the implementation effort for the event distribution pattern could be significantly decreased by reusing the events resulting from the architecture principles of CQRS. This solution described by the first expert shows an advantage of the event distribution pattern compared to the component interceptor pattern in specific cases where the existing application architecture is based on events. This presents an advantage in specific implementation case.

Chapter 10

Conclusion

The research presented in this thesis makes a contribution to the area of runtime variability in business software as a service by presenting four patterns for two areas of variability together with an analysis and expert review of these patterns. We will review our results and conclusions per research question, starting with the first question:

> 1. How can software design patterns for runtime variability be evaluated in the context of a multi-tenant business application?

This research question forms the basis for our pattern descriptions and evaluation. To evaluate patterns we must first be able to capture and describe those patterns. Subquestion 1.a is set to determine which description methods are applicable:

> 1.a What description methods are applicable to software design patterns?

We compared UML methods developed by Evitts (2000) for pattern description, Design Pattern Modeling Language by Mapelsden et al. (2002) and formal approachs such as a CASE based technique described by Lauder & Kent (1998). These methods focussed on creating descriptions suitable for object oriended design patterns. The application of software patterns to the description of multi-tenant systems is different from a pure object oriented design pattern. An object oriented design pattern describes common solutions to problems in object oriented software design. The most important difference between object oriented 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.

To accomodate the descriptions of these system level patterns we described an approach for modeling patterns at different levels in section 4.2, based on the concepts of pattern descriptions by Lauder & Kent (1998). For our further research questions we required the ability to compare patterns on different properties. To allow for patterns to be compared we must first establish that all patterns involved share the same functional goal.

The subsequent part of the first research question are the criteria with which the patterns must be evaluated. Subquestion 1.b was set to determine these criteria:

> 1.b What are evaluation criteria for software design patterns in the context of business software?

Evaluating patterns on criteria selected as relevant from established literature ensures the results of our evaluation are applicable to the envisioned users of the patterns. We did not set criteria for the pattern descriptions on clarity, this has been done by Hsueh et al. (2008). Instead we determined the quality attributes Security, Performance, Scalability, Maintainability and Implementation effort based on studies of SaaS adoption and the risks and opportunities by Benlian & Hess (2011) and Wu et al. (2011).

The second research question set the bounds for selection and description of patterns from our case studies.

> 2. How can a multi-tenant business application provide runtime variability on datamodel and application behaviour?

The selection of patterns was conducted in the areas of datamodel variability and application behaviour variability as two separate subquestions.

> 2.a What software design patterns are applicable to runtime variability of the datamodel?
>
> 2.b What software design patterns are applicable to runtime variability of application behaviour?

We selected two patterns for each area. Patterns applicable to runtime variability of the datamodel are the datasource router pattern we observed in a multi-tenant business reporting application and the custom property objects pattern we observed in a SaaS based CRM application for small to medium sized enterprises. We provided pattern descriptions of both patterns in chapter 6. Both patterns adhere to a single functional model as required by the description technique. This functional model is included in chapter 6.

On the subject of datamodel extension we found two applicable patterns, the component interceptor pattern which we observed in a logistical planning system and the event distribution pattern which we observed in a webbased application for the support of medical staff. We providede pattern descriptions for both patterns along with the shared functional model in chapter 7.

The final research question brings the patterns and evaluation criteria together in our analysis and subsequent expert review in chapters 8 and 9:

> 3. How do the selected patterns compare on evaluation criteria as selected in research question 1?

The analysis compared patterns on all selected quality attributes, considering their relative merits for each property independently. This analysis was followed by the expert review to ensure the analysis matched the views of the experts and to provide the target audience with expert comments on the risks and opportunities for each pattern. The experts provided input on the differences between the patterns and practical implications of implementing the patterns in different types of architectures.

For datamodel extension patterns 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 datamodel avoiding the need to test every possible variation when adapting the software.

Comparing the patterns for adding behaviour we conclude that the component interceptor pattern has advantages on performance, because code is executed immediately within the application avoiding any overhead in network transport and the need for additional resources. The component interceptor pattern also requires less implementation effort because no external systems need to be deployed. The event distribution pattern on the other hand holds advantages in scalability and maintainability by executing the custom behaviour outside of the application. This separation allows for better scalability by independently scaling the extension components from the rest of the application and increases maintainability because there is no dependency from the application on the extension components.

10.1 Future Research

Future research into the weighting of the quality attributes based on the implementation context could provide valuable practical methods for businesses implementing these patterns. The description methods we designed provide the groundwork for the development of a pattern library with patterns applicable to the areas of datamodel extension and the addition of behaviour as well as the description of patterns in other areas not covered in this research. Further research into pattern evaluation is necessary to develop a quantitative method to evaluate the resulting architectures allowing for the comparison of much larger volumes of patterns compared to the expert reviews conducted in this research.

References

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

Bayer, J., Gerard, S., Haugen, O., Mansell, J., Mller-Pedersen, B., Oldevik, J., Tessier, P., Thibault, J.-P., & Widen, T. (2006). Consolidated product line variability modeling.

Benlian, A., & Hess, T. (2011). Opportunities and risks of software-as-a-service: Findings from a survey of it executives. *Decision Support Systems*, *52*(1), 232 – 246.

Bezemer, C.-P., & Zaidman, A. (2010). Multi-tenant saas applications: maintenance dream or nightmare? In *Proceedings of the Joint ERCIM Workshop on Software Evolution and International Workshop on Principles of Software Evolution*, (pp. 88–92). New York, USA: ACM.

Carpenter, B., Fox, G., Ko, S. H., & Lim, S. (1999). Object serialization for marshalling data in a java interface to mpi. In *Proceedings of the ACM 1999 conference on Java Grande*, (pp. 66–71). New York, NY, USA: ACM.

Chen, P. P.-S. (1976). The entity-relationship modeltoward a unified view of data. *ACM Trans. Database Syst.*, *1*(1), 9–36.

Cook, D., T.D. Campbell (1979). *Quasi-Experimentation: Design and Analysis Issues for Field Settings*. Houghton Mifflin.

Dubey, A., & Wagle, D. (2007). Delivering software as a service. *The McKinsey Quarterly*, (pp. 1–12).

Espadas, J., Molina, A., Jimnez, G., Molina, M., Ramrez, R., & Concha, D. (2011). A tenant-based resource allocation model for scaling software-as-a-service applications over cloud computing infrastructures. *Future Generation Computer Systems*, (0).

Evans (2003). *Domain-Driven Design: Tacking Complexity In the Heart of Software*. Boston, MA, USA: Addison-Wesley.

Evitts, P. (2000). *A UML Pattern Language*. Thousand Oaks, CA, USA: New Riders Publishing.

Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Boston, MA, USA: Addison-Wesley.

Fowler, M. (2003). Patterns. *Software, IEEE*, *20*(2), 56 – 57.

France, R., Kim, D.-K., Ghosh, S., & Song, E. (2004). A uml-based pattern specification technique. *Software Engineering, IEEE Transactions on*, *30*(3), 193 – 206.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1993). Design patterns: Abstraction and reuse of object-oriented design. In O. Nierstrasz (Ed.) *ECOOP 93 Object-Oriented Programming*, vol. 707 of *Lecture Notes in Computer Science*, (pp. 406–431). Springer Berlin / Heidelberg.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*. Boston, USA: Addison-Wesley Longman Publishing Co.

Gomaa, H., & Hussein, M. (2007). Model-based software design and adaptation. In *Proceedings of the 2007 International Workshop on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '07, (pp. 7–). Washington, DC, USA: IEEE Computer Society.
URL `http://dx.doi.org/10.1109/SEAMS.2007.13`

Google (2008). Protocol buffers - google's data interchange format. `http://code.google.com/p/protobuf/`. Accessed: 27-01-2013.

Grotzke, M., Sweet, N., & Levenstein, R. (2009). Kryo - fast, efficient java serialization and cloning. `http://code.google.com/p/kryo/`. Accessed: 27-01-2013.

Guo, C. J., Sun, W., Huang, Y., Wang, Z. H., & Gao, B. (2007). A framework for native multi-tenancy application development and management. *E-Commerce Technology, IEEE International Conference on, and Enterprise Computing, E-Commerce, and E-Services, IEEE International Conference on*, *0*, 551–558.

Hsueh, N.-L., Chu, P.-H., & Chu, W. (2008). A quantitative approach for evaluating the quality of design patterns. *Journal of Systems and Software*, *81*, 1430–1439.

International Organization for Standardisation (1991). Iso/iec: 9126 information technology-software product evaluation-quality characteristics and guidelines for their use.

Jansen, S., Houben, G.-J., & Brinkkemper, S. (2010). Customization realization in multi-tenant web applications: Case studies from the library sector. In B. Benatallah, F. Casati, G. Kappel, & G. Rossi (Eds.) *Web Engineering*, vol. 6189 of *Lecture Notes in Computer Science*, (pp. 445–459). Springer Berlin / Heidelberg.

Jaring, M., & Bosch, J. (2002). Representing variability in software product lines: A case study. In G. Chastek (Ed.) *Software Product Lines*, vol. 2379 of *Lecture Notes in Computer Science*, (pp. 219–245). Springer Berlin / Heidelberg.

Jung, H.-W., Kim, S.-G., & shin Chung, C. (2004). Measuring software product quality: a survey of iso/iec 9126. *Software, IEEE*, *21*(5), 88 – 92.

Kazman, R., Klein, M., Barbacci, M., Longstaff, T., Lipson, H., & Carriere, J. (1998). The architecture tradeoff analysis method. *Engineering of Complex Computer Systems, IEEE International Conference on*, *0*, 68.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., & Irwin, J. (1997). Aspect-oriented programming. In M. Akit, & S. Matsuoka (Eds.) *ECOOP'97 Object-Oriented Programming*, vol. 1241 of *Lecture Notes in Computer Science*, (pp. 220–242). Springer Berlin Heidelberg.

Kitchenham, B., & Pfleeger, S. (1996). Software quality: the elusive target [special issues section]. *Software, IEEE*, *13*(1), 12 –21.

Lauder, A., & Kent, S. (1998). Precise visual specification of design patterns. In E. Jul (Ed.) *ECOOP98 Object-Oriented Programming*, vol. 1445 of *Lecture Notes in Computer Science*, (pp. 114–134). Springer Berlin / Heidelberg.

Mapelsden, D., Hosking, J., & Grundy, J. (2002). Design pattern modelling and instantiation using dpml. In *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for internet, mobile and embedded applications*, CRPIT '02, (pp. 3–11). Darlinghurst, Australia: Australian Computer Society, Inc.

Mapelsden, D., Hosking, J., & Grundy, J. (2007). A visual language for design pattern modelling and instantiation. In T. Taibi (Ed.) *Design Pattern Formalization Techniques*, (pp. 20–43).

Marston, S., Li, Z., Bandyopadhyay, S., Zhang, J., & Ghalsasi, A. (2011). Cloud computing the business perspective. *Decision Support Systems*, *51*(1), 176 – 189.

Mietzner, R., Metzger, A., Leymann, F., & Pohl, K. (2009). Variability modeling to support customization and deployment of multi-tenant-aware software as a service applications. In *Proceedings of the 2009 ICSE Workshop on Principles of Engineering Service Oriented Systems*, (pp. 18–25). Washington, DC, USA: IEEE Computer Society.

Montero, I., Pena, J., & Ruiz-Cortes, A. (2008). Representing runtime variability in business-driven development systems. In *Composition-Based Software Systems, 2008. ICCBSS 2008. Seventh International Conference on*, (pp. 228 –231).

Nussbaum, D., & Agarwal, A. (1991). Scalability of parallel machines. *Commun. ACM*, *34*(3), 57–61.

Ohta, K., & Colebourne, S. (2011). Messagepack format specification. `http://wiki.msgpack.org/display/MSGPACK/Format+specification`. Accessed: 27-01-2013.

Oracle.com (2008). Design question: Self join.
URL `http://asktom.oracle.com/pls/asktom/f?p=100:11:0::::P11_QUESTION_ID:1389366900346164087`

Rosengard, J.-M., & Ursu, M. (2004). Ontological representations of software patterns. In M. Negoita, R. Howlett, & L. Jain (Eds.) *Knowledge-Based Intelligent Information and Engineering Systems*, vol. 3215 of *Lecture Notes in Computer Science*, (pp. 31–37). Springer Berlin Heidelberg.

Rumbaugh, J., Jacobson, R., & Booch, G. (1999). *The Unified Modelling Language Reference Manual*. Addison-Wesley, 1 ed.

Slaughter, S. A., Harter, D. E., & Krishnan, M. S. (1998). Evaluating the cost of software quality. *Commun. ACM*, *41*(8), 67–73.
URL `http://doi.acm.org/10.1145/280324.280335`

Subashini, S., & Kavitha, V. (2011). A survey on security issues in service delivery models of cloud computing. *Journal of Network and Computer Applications*, *34*(1), 1 – 11.

Sun, W., Zhang, X., Guo, C. J., Sun, P., & Su, H. (2008). Software as a service: Configuration and customization perspectives. In *Proceedings of the 2008 IEEE Congress on Services Part II*, (pp. 18–25). Washington, DC, USA: IEEE Computer Society.

Svahnberg, M., van Gurp, J., & Bosch, J. (2005). A taxonomy of variability realization techniques. *Software: Practice and Experience*, *35*(8), 705–754.

Tellis, W. (1997). Introduction to case study. *The qualitative report*, *3*(2).

Thekkath, C. A., & Levy, H. M. (1993). Limits to low-latency communication on high-speed networks. *ACM Trans. Comput. Syst.*, *11*(2), 179–203.

Turner, M., Budgen, D., & Brereton, P. (2003). Turning software into a service. *Computer*, *36*(10), 38–44.

Vaishnavi, V. K., & Kuechler, W., Jr. (2007). *Design Science Research Methods and Patterns: Innovating Information and Communication Technology*. Boston, MA, USA: Auerbach Publications, 1st ed.

Weinreich, R., Ziebermayr, T., & Draheim, D. (2007). A versioning model for enterprise services. In *Proceedings of the 21st International Conference on Advanced Information Networking and Applications Workshops - Volume 02*, AINAW '07, (pp. 570–575). Washington, DC, USA: IEEE Computer Society.

Wu, W.-W., Lan, L. W., & Lee, Y.-T. (2011). Exploring decisive factors affecting an organization's saas adoption: A case study. *International Journal of Information Management*, *31*(6), 556 – 563.

Yin, R. K. (2002). *Case Study Research : Design and Methods*. Applied Social Research Methods. SAGE Publications, 3rd ed.

Young, G. (2013). Event centric: Finding simplicity in complex systems. (Unpublished).