UTRECHT UNIVERSITY

Variability in Multi-Tenant Enterprise Software

by

Jaap Kabbedijk

A thesis submitted in partial fulfillment for the
degree of Doctor

in the

Faculty of Science
Department of Information and Computing Sciences

December 2014

SIKS Dissertation Series No. 2014-29
The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

The cover image is from the board game Tzolkin: The Mayan Calendar by Czech Games Edition. The photo is provided by BoardGameGeek member Joshua R.

ISBN/EAN: 978-90-393-6177-1

Variability in Multi-Tenant Enterprise Software

Variabiliteit in Multi-tenant Bedrijfssoftware
(met een samenvatting in het Nederlands)

Proefschrift

ter verkrijging van de graad van doctor aan de Universiteit Utrecht op gezag van de rector magnificus, prof. dr. G.J. van der Zwaan, ingevolge het besluit van het college voor promoties in het openbaar te verdedigen op dinsdag 23 december 2014 des middags te 12.45 uur door

Jaap Kabbedijk

geboren op 27 mei 1986 te Oosterhout

Promotor:	Prof.dr. S. Brinkkemper
	Prof.dr.ir. J.C. Wortmann
Copromotor:	Dr. R.L. Jansen

This research was financially supported by the NWO ‘Product as a Service’ project.

"Somewhere in the deeply remote past it seriously traumatized a small random group of atoms drifting through the empty sterility of space and made them cling together in the most extraordinarily unlikely patterns. These patterns quickly learnt to copy themselves (this was part of what was so extraordinary of the patterns) and went on to cause massive trouble on every planet they drifted on to. That was how life began in the Universe."

Douglas Adams, The Hitchhiker's Guide to the Galaxy

Preface

Before I started my PhD., I was not really aware of what such an endeavour would entail. Now, almost four years later, I do know. I now know one can not fulfil this journey alone. The are many people to thank and I will try to do this here. Most probably I missed the most obvious, important and valuable ones. For this I apologize beforehand. If you are one of them, feel blessed; you can count yourself among my most important and valuable friends.

Right after my first publication and presentation at a scientific conference, I decided the thrill and travel of academic interaction was something I would like to experience more. This was during my master studies and I made the life changing choice to become a PhD. This was a major choice. An important choice, and a choice I don't regret.

Using the enthusiasm I had when I started, the literal and intellectual travels began. On the way I learned to value the people supporting my professionally. Slinger, thank you for helping me structure and improve my studies. Thanks also for not only being an academic mentor, but also a friend for beers and laughs.

Unsurprisingly, my gratitude also goes to my promotor Sjaak, who was busy at times, but always tried to help me find out what was best for me. I also express my thanks to my second promotor Hans Wortmann and the people I worked with during my research project and at the many software companies I visited. There are too many people to name them all, but Liz, Werner, Ronald, Rolf and Machiel, you are certainly among the many I should actually all thank explicitly.

So, now for the people that helped me out, often not by supporting me directly in writing my thesis, but by being there for and supporting me. Of course my parents; pap en mam, bedankt dat jullie er altijd voor me zijn. Also, all my friends. There are, luckily, too many of you to write all the names down, but thank you all. Special thanks goes to Kevin, who has been my soul mate for almost ten years now. Last, but certainly not least, my love goes to Eline.

Knowing what I know now, I can safely state that I learned a lot about software patterns, software architecture, research methods and the academic culture. But most importantly, I learned that a PhD., like most things in life, is enjoyed best when you have the people you love, support you.

Jaap

Contents

Preface **iv**

1 Introduction **3**
1.1 Motivation . . . 3
1.2 Scientific Relevance . . . 5
1.3 Positioning the Research . . . 6
1.3.1 Enterprise Software . . . 7
1.3.2 Software as a Service . . . 8
1.3.3 Software Architecture . . . 10
1.3.4 Software Quality . . . 10
1.3.5 Software Patterns . . . 12
1.3.6 Variability . . . 13
1.4 Research Approach . . . 13
1.4.1 Research Questions . . . 14
1.4.2 Research Methods . . . 17
1.4.3 Validation and Evaluation . . . 20
1.5 Dissertation Outline . . . 21

I Variability and Multi-tenancy **25**

2 Defining Multi-Tenancy **27**
2.1 Introduction . . . 28
2.2 Research Method . . . 30
2.2.1 Academic Literature Collection . . . 31
2.2.2 Industrial Literature Collection . . . 32
2.3 Classification . . . 34
2.3.1 Academic Literature Classification . . . 34
2.3.2 Industrial Literature Classification . . . 36
2.4 Observations . . . 37
2.4.1 Academic Paper Results . . . 38
2.4.2 Blog Post Results . . . 38
2.5 Definition . . . 40
2.6 Research Agenda . . . 43
2.7 Threats to Validity . . . 46

2.8 Conclusion 46

3 The Role of Variability Patterns 49
3.1 Introduction 50
3.2 Concepts 51
3.2.1 Tenant-based Variability 51
3.2.2 Variability Patterns 52
3.3 Conceptual Model 53
3.3.1 Application Example 54
3.4 Discussion 55
3.5 Conclusion 56

4 Variability in Multi-tenant Systems 57
4.1 Introduction 58
4.2 Research Approach 59
4.2.1 Validation 59
4.3 Related Work and Definitions 60
4.3.1 Multi-tenancy 60
4.3.2 Variability 61
4.3.3 Software Patterns 62
4.4 User-Variability Trade-off 63
4.5 Variability Patterns 64
4.5.1 Customizable Data Views 64
4.5.2 Module Dependent Menu 65
4.5.3 Pre/Post Update Hooks 67
4.6 Conclusion and Future Research 68

5 Variability Consequences of the CQRS Pattern 69
5.1 Introduction 70
5.2 Related Work 71
5.3 Research Approach 73
5.3.1 Research Questions 74
5.3.2 Validation 75
5.4 CQRS Implementation 75
5.5 CQRS Sub Patterns 77
5.5.1 Event Sourcing 77
5.5.2 Event Store 77
5.5.3 Aggregate Root 78
5.5.4 Command Handler 79
5.5.5 Query Model Builder 80
5.5.6 Query Handler 80
5.5.7 Snapshotting 81
5.6 Variability Influences 82
5.7 Discussion and Future Research 83
5.8 Conclusion 84

II Selecting Patterns in Systems Design 85

6 Multi-Tenant Architecture Assessment 87
6.1 Introduction . . . 88
6.2 Research Approach . . . 89
6.2.1 Structured Literature Research . . . 90
6.2.2 Expert Validation . . . 92
6.3 Multi-Tenant Architecture Assessment Model . . . 93
6.4 Multi-tenant Architectures . . . 94
6.4.1 Expert Validation . . . 97
6.5 MTA Assessment Criteria . . . 98
6.5.1 Expert Evaluation . . . 100
6.6 MTA Decision Matrix . . . 101
6.7 Discussion and Conclusion . . . 104

7 Comparing Dynamical Adaptation Patterns 105
7.1 Introduction . . . 106
7.2 Related Work . . . 107
7.3 Research Approach . . . 110
7.3.1 Validation . . . 111
7.4 Pattern Description Method . . . 111
7.5 Dynamic Functionality Adaptation Patterns . . . 114
7.5.1 Problem Statement . . . 114
7.5.2 Component Interceptor Pattern . . . 115
7.5.3 Event Distribution Pattern . . . 116
7.5.4 Pattern Comparison . . . 118
7.6 Dynamic Data Model Extension Patterns . . . 122
7.6.1 Problem Statement . . . 122
7.6.2 Datasource Router Pattern . . . 124
7.6.3 Custom Property Object Pattern . . . 126
7.6.4 Pattern Comparison . . . 127
7.7 Conclusion . . . 132

8 Software Pattern Evaluation Method 135
8.1 Introduction . . . 136
8.2 Related Work . . . 137
8.3 Research Approach . . . 138
8.4 SPEM - Software Pattern Evaluation Method . . . 141
8.5 Method Evolution . . . 144
8.5.1 Initial Method Construction . . . 144
8.5.2 Method Evaluation . . . 146
8.6 SPEM Impementation . . . 150
8.7 Conclusion . . . 151

9 Conclusion 155
9.1 Contributions and Evaluations . . . 156

9.2 Implications . . . 164
9.3 Reflection . . . 166
9.4 Limitations and Future Research . . . 169

Bibliography **173**

A Pattern Catalogue **185**
Customizable Data Views Pattern . . . 186
Module Dependent Menu Pattern . . . 188
Pre/Post Update Hooks Pattern . . . 190
CQRS Pattern . . . 192
Event Sourcing Pattern . . . 194
Event Store Pattern . . . 195
Aggregate Root Pattern . . . 196
Command Handler Pattern . . . 197
Query Manager Pattern . . . 198
Snapshotting Pattern . . . 200
Dedicated Application and Database Server Pattern . . . 201
Shared Application Server / Dedicated Database Server Pattern . . . 203
Shared Instance / Dedicated Database Server Pattern . . . 205
Dedicated Application Server / Shared Database Server Pattern . . . 207
Shared Application and Database Server Pattern . . . 209
Shared Instance and Database Server Pattern . . . 211
Dedicated Application Instance / Shared Database Pattern . . . 213
Shared Application Server and Database Pattern . . . 215
Shared Instance and Database Pattern . . . 217
Dedicated Application Server / Shared Schema Pattern . . . 219
Shared Application Server and Database Schema Pattern . . . 221
Shared Instance and Database Schema Pattern . . . 223
Component Interceptor Pattern . . . 225
Event Distribution Pattern . . . 227
Datasource Router Pattern . . . 229
Custom Property Object Pattern . . . 231

B Publications used in the Structured Mapping Study **233**

C List of Acronyms **239**

D Personal Publication List **241**

E Summary **245**

F Samenvatting **249**

G SIKS Dissertation Series **253**

Introduction

Chapter 1

Introduction

1.1 Motivation

Enterprise software is the backbone of many companies, both large and small, for many years already (Engelstätter, 2012). Different types of enterprise software exist, ranging from content management systems to customer relationship management products, to large Enterprise Resource Planning (ERP) applications (Hendricks, Singhal, and Stratman, 2007). Traditionally, such software applications were deployed on-premises at the customer's site. This means the customer is responsible for the infrastructure and maintenance contracts in order to be able to use the product in a reliable way. This made enterprise software deployment and maintenance a costly endeavour, with long lead times. The upside of on-premises deployments, however was the opportunity of custom functionality per customer. Starting in the 1990s, enterprise software applications were increasingly offered to companies as hosted solutions (Hoch, Kerr, Griffith, et al., 2001), i.e. Application Service Providers (ASPs). Application service providers hosted the application on behalf of companies, leading to lower maintenance and infrastructure fees for customers. For the software vendor this means a separate installation of the software has to be deployed for every customer, even if multiple customers use a similar product.

Gradually the ASP deployment model changed to a more centralized deployment model; Software as a Service (SaaS) (Turner, Budgen, and Brereton, 2003). Within the SaaS paradigm, a single instance of a software product may serve many different customers. The sharing of resources has many benefits, among which lower costs for both provider and customer and convenient ways of sharing data (Wu,

Lan, and Lee, 2011), but limited the options of custom functionality which was one of the big benefits of on-premises software. SaaS is often seen as a part of the *cloud computing* paradigm, which is a term for describing the offering of software, hardware or data through the internet (Armbrust et al., 2010). Sharing resources by offering a shared application or database instance and providing the service through the internet, is also referred to as *multi-tenancy* (Bezemer and Zaidman, 2010), in which the lack of customizability is counteracted by offering a certain level of variability. Multi-tenancy is defined as "a property of a system where multiple customers, so-called tenants, transparently share the system's resources, such as services, applications, databases, or hardware, with the aim of lowering costs, while still being able to exclusively configure the system to the needs of the tenant" (Kabbedijk, Bezemer, Jansen, and Zaidman, 2014 (In Press)).

Currently, SaaS has become the de facto standard for enterprise software applications. Enterprise software which provides functionality for a specific business process, such as Content Management Systems (CMSs) (e.g. Wordpress) and Customer Relationship Management (CRM) systems (e.g. Salesforce) are already primarily provided as Software as a Service (SaaS) offerings, but more complex software, e.g. ERP products are still lacking (Forrester, 2013). Currently, just 2% of large and middle sized companies have adopted ERP as an online product while almost half of all companies is willing to change from an on-premises to an online deployment (Forbes, 2014). Enterprise software vendors indicate their current goal is to migrate their products to a SaaS deployment model in the near future (Forrester, 2012), but are struggling to comply. When non-trivial business processes, such as planning or financial management are concerned, customers' requirements differ between each other. This means complex enterprise software needs to comply to many different customer requirements, leading to a need for variability in functionality. The lack of multi-tenant offerings of complex enterprise software, such as ERP systems, is an indicator software companies struggle to offer variability in multi-tenant enterprise software.

> **Problem Statement** — It is unclear to software producing organizations how variability must be implemented in multi-tenant enterprise software. This uncertainty hampers the production and adoption of online enterprise software.

This nescience of software producing organizations inhibits the creation of multi-tenant enterprise software and limits the functionality of software offered to customers. In this dissertation, we pursue methods, tools, and software patterns for the realization of multi-tenancy in multi-tenant enterprise software.

1.2 Scientific Relevance

In order to offer different customers different functionality in their software products, variability mechanisms are often applied among the different products from a software vendor (Pohl, Böckle, and Linden, 2005). Variability has always been a significant research area in Software Product Lines (SPLs) (Van Gurp, Bosch, and Svahnberg, 2001). The emphasis in SPLs is on the manufacturing of different products all based on the same shared set of assets, as is common in, for instance, the automotive industry and the manufacturing of mobile handsets (Thiel and Hein, 2002). Variability the whole software development life cycle (Michalik, Avgeriou, Tofan, Galster, and Weyns, 2014) and binding to a particular variant of a product can occur at different moments life cycle, e.g. at run-time (Svahnberg, Gurp, and Bosch, 2005).

The level of variability in software products is strongly dependent on not only the architecture of the product, but also the architecture of the platform or operating system the product runs on. Implementing localization in iOS and Android applications, for example, is a seemingly simple variability issue, which can be really hard to tackle, due to the lack of support for localization in both operating systems (Galster, Männistö, Weyns, and Avgeriou, 2014). The concept of variability is closely tied to software architecture, since the level of variability in a software product is dependent on variability choices made early on in the architecting process. Variability also affects many software architecture quality attributes and can even be seen as a quality attribute itself, amplifying the strong link between variability and software architecture (Galster, Männistö, Weyns, and Avgeriou, 2014). Most concepts in variability research are currently geared towards on-premises software and fail to address variability in online software. The concept of *run-time variability* can be used the address the problem of varying requirements while a software product is running, but is based on single-tenant software instead of multi-tenant applications. In this dissertation, the problem of implementing variability in multi-tenant software is addressed.

The architecting process in software engineering is frequently documented and communicated in the form of software patterns (Gamma, Helm, Johnson, and Vlissides, 1995). Depending on the pattern description form (e.g. Gang of Four (GoF) based form (Gamma, Helm, Johnson, and Vlissides, 1995) or Pattern-Oriented Software Architecture (POSA) based form (Buschmann, Meunier, Rohnert, Sommerlad, and Stal, 1996)), patterns describe not only a proposed solution for a recurring problem, but also consequences of applying the pattern. It is of importance to know the consequences of applying a pattern, and the effect on software quality, before implementing a pattern. In software architecture, the quality of a system is assessed by evaluating quality attributes, as proposed by ISO/IEC 25010 (ISO/IEC, 2011) (previously ISO/IEC 9126 (ISO/IEC, 2001)), of a specific system implementation (Bass, Clements, and Kazman, 2013). Patterns, by definition, are general solutions for a problem instead of a specific implementation (Coplien and Alexander, 1996). Evaluating the effect of applying a *general solution* in depth, instead a *specific implementation* is currently lacking in scientific literature. Software architecture quality attributes are often measured using metrics of the system implementation (Kan, 2002). When evaluating patterns, this is not possible and should be performed in a different way. This dissertation proposes a method for using expert focus groups (Kitzinger, 1995) to evaluate to software quality affected by the software pattern.

The two leading conferences in software architecture research (i.e. the European Conference on Software Architecture (ECSA) and the Working IEEE/IFIP Conference on Software Architecture (WICSA) call explicitly for studies on *software architectures for cloud architectures*, *architectural patterns* and *quality attributes*, in their calls for papers, indicating the active research field worldwide. Additionally, the Pattern Languages of Programs (PLoP) community expresses the need for pattern creation in their call for paper for academics and practitioners, in order to guide the domain of software engineering. This research aims at answering these calls by adding to the body of knowledge of software architecture research and developing a software pattern catalogue.

1.3 Positioning the Research

This section discusses the important concepts used within this thesis. For every concept relevant literature is discussed, after which a definition of the concept is provided.

1.3.1 Enterprise Software

The software business is often characterized to concern *software products* and *software services* (Cusumano, 2004). For product companies, most of the firm's revenues come from selling commercial software or licenses. Service companies earn most of the revenues through selling additional services, such as customizations and support. The difference between the two different types, however, is not clear-cut, and many hybrid companies exist (Nambisan, 2001). An important difference between the software business and other businesses lies in the fact that its main product is intangible and has a variable production cost of near zero (Messerschmitt and Szyperski, 2005). Focusing on software products, enterprise software serves the needs of companies, instead of individual users and has become the de-facto software used in large organizations (Brown and Vessey, 2003). Types of products often mentioned as examples of enterprise software include Enterprise Resource Planning (ERP) software, Content Management Systems (CMSs) and Customer Relationship Management (CRM) applications (Hendricks, Singhal, and Stratman, 2007). An overview of different types of enterprise software has been given by Xu and Brinkkemper (2007), who define the following four types of software:

- Shrink-wrapped software — Software that is boxed, shrink-wrapped and sold in stores.
- Commercial Off-The-Shelf (COTS) software — "Software sold, leased, or licensed to the general public; offered by a vendor trying to profit from it; supported and evolved by the vendor, who retains the intellectual property rights; available in multiple, identical copies and used without source code modification." (Brownsword, Oberndorf, and Sledge, 2000, p.49).
- Packaged software — Software products obtainable from software vendors, which generally little modification or customization.
- Commercial Software — Software which can be purchased through the retail market.

Enterprise software is also called 'enterprise applications' or 'business software' (Swanson and Wang, 2005). In this dissertation the term 'software product' is used, which refers to packaged enterprise software. Currently, the delivery of enterprise

software is shifting from an on-premises deployment model to an online deployment model (D'souza, Kabbedijk, Seo, Jansen, and Brinkkemper, 2012), which will be discussed further in the next section.

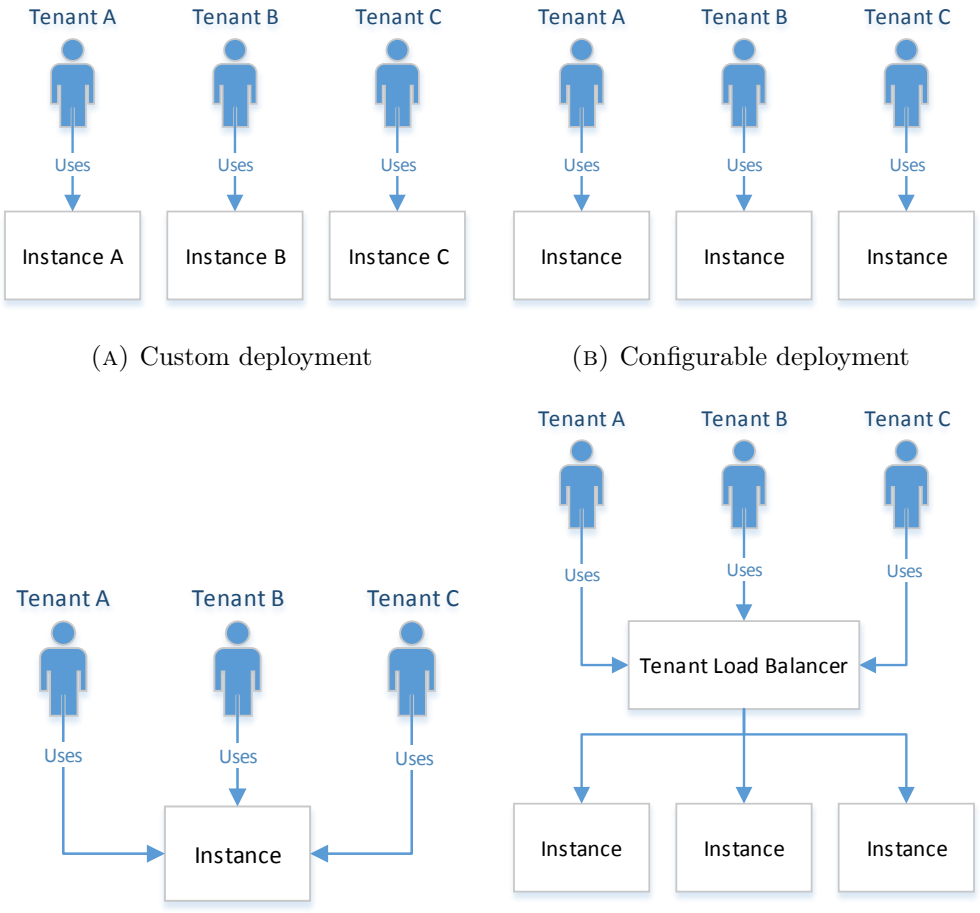
Definition of Enterprise Software — "Enterprise applications are about the display, manipulation, and storage of large amounts of often complex data and the support or automation of business processes with that data." (Fowler, 2003, p. xviii)

1.3.2 Software as a Service

The concept of Software as a Service (SaaS) was first mentioned in an internal document from the Software & Information Industry Association (SIIA), in which it was introduced as the successor of Application Service Provider (ASP) (Hoch, Kerr, Griffith, et al., 2001). Turner, Budgen, and Brereton (2003) later stated that SaaS "focusses on separating the possession and ownership of software from its use". Typically, SaaS is delivered as a set of services that can be accessed and use through the internet (Dubey and Wagle, 2007). The service-oriented approach in SaaS allows for much more flexibility in system engineering, as opposed to the on-premises design (Gold, Mohan, Knight, and Munro, 2004). Four different maturity levels of online software deployment exist according to Chong and Carraro (2006), which are shown in Figure 1.1.

A deployment model in which all tenants have a custom deployment, which is hosted by a hosting provider is shown in Figure 1.1a. This level is similar to the earlier mentioned ASP model (Tao, 2001). This deployment model allows for a high level of customization but lacks the ability to scale well horizontally (Arlitt, Krishnamurthy, and Rolia, 2001) if a provider gets a lot of tenants. Also, since all instances are different, updating the software can be cumbersome. The configurable deployment model (cf. Figure 1.1b) offers all tenants a similar customizable instance of a software product. This improves the ease of software updating, while keeping the possibility to customize the software product. This model, however, similar to the previous model, does not scale well if the customer base grows.

One of the main potential benefits of SaaS is the sharing of resources among tenants (Dubey and Wagle, 2007). The configurable shared deployment model (cf. Figure 1.1c), in which one instance of a software product is shared among all tenants. This approach allows for a high level of maintainability (Perepletchikov, Ryan, Frampton, and Tari, 2007), but limits the levels of scalability and functional



(A) Custom deployment

(B) Configurable deployment

(C) Configurable, shared deployment

(D) Configurable, scalable deployment

FIGURE 1.1: SaaS Maturity Levels

variability(Van Gurp, Bosch, and Svahnberg, 2001). The final level of SaaS maturity is shown in Figure 1.1d. In this model, a load balancer is used to balance the tenant requests among the different similar product instances, leading to a high level of scalability (Rimal, Choi, and Lumb, 2009).

SaaS is often characterized as being the top level in the online software stack (also called "cloud software" (Mell and Grance, 2011)). The other layers are:

- Platform as a Service (PaaS) — Online provision of the software platform where applications run on (Vaquero, Rodero-Merino, Caceres, and Lindner, 2008).
- Infrastructure as a Service (IAAS) — Provision model for the sourcing of the infrastructure over the internet (Bhardwaj, Jain, and Jain, 2010).

Definition of SaaS — "The capability provided to the consumer is to use the provider's applications running on a cloud infrastructure. The applications are accessible from various client devices through either a thin client interface, such as a web browser (e.g., web - based email) or a program interface. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, storage, or even individual application capabilities, with the possible exception of limited user - specific application configuration settings." (Mell and Grance, 2011, p. 2)

1.3.3 Software Architecture

The concept of software architecture originates from the 1960s but was made popular by Shaw and Garlan (1996). Software architecture is a discipline that primarily consists of three different practices:

- The structure of a software system, i.e. its components and relationships (Clements, Garlan, Bass, Stafford, Nord, Ivers, and Little, 2002).
- The process of selecting the right architecture; referred to in this dissertation as 'architecting process'.
- The documentation of a software system and the decisions made concerning the system (Bass, Clements, and Kazman, 2013).

The focus in this dissertation is on the structure of software systems. Designing the structure of an architecture, description of the architecture can be done in different ways. Taylor, Medvidovic, and Dashofy (2010) state that architectural patterns are a suitable way to model domain knowledge on application structures.

Definition of Software Architecture — "The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them." (Bass, Clements, and Kazman, 2013, p. 30)

1.3.4 Software Quality

Different views exist on what software quality entails (Kitchenham and Pfleeger, 1996). A dominant view on software quality is the user perspective, which focusses

on how *users* experience the quality of a product. In this dissertation, we focus more on the product perspective, which puts attributes of the *product* central instead of the subjective perspective of the user.

The quality of software is often expressed as a fixed list of attributes, defined in the ISO/IEC 25010 standard (ISO/IEC, 2011). In this standard the following main attributes are expressed:

- **Performance efficiency** — Degree to which the software product provides appropriate performance, relative to the amount of resources used, under stated conditions.
- **Compatibility** — The ability of multiple software components to exchange information or to perform their required functions while sharing the same environment.
- **Usability** — Degree to which the software product can be understood, learned and used, when used under specified conditions.
- **Reliability** — Degree to which the software product can maintain a specified level of performance when used under specified conditions.
- **Security** — The protection of system items from accidental or malicious access, use, modification, destruction, or disclosure.
- **Maintainability** — Degree to which the software product can be modified. Modifications may include corrections, improvements or adaptation of the software to changes in the environment, and in requirements and functional specifications.
- **Portability** — Degree to which the software product can be transferred from one environment to another

The attributes mentioned above, all consist of sub-attributes (e.g. reusability or changeability), which will not be discussed in depth in the dissertation. Determining the quality attributes of a software system is done by measuring proxies within the system, such as the response time or total lines of code (Kan, 2002).

Definition of Software Quality — "Conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software." (Pressman, 1994, p. 388)

1.3.5 Software Patterns

Patterns are originally introduced by Alexander, Ishikawa, Silverstein, Jacobson, Fiksdahl-King, and Angel (1977), who wrote a pattern language to describe the design of a city. The concept of describing common problems in a design and bundling them with a generic solution was picked up in the software engineering domain and popularized by Gamma, Helm, Johnson, and Vlissides (1995). They present 23 patterns (e.g. Observer Pattern, Abstract Factory), which all describe a specific object oriented design problem by listing the following attributes: 1. Name, 2. Intent, 3. Motivation, 4. Applicability, 5. Structure, 6. Consequences and 7. Known Uses. By using the same description format for all patterns, they became usable references for software engineers in designing software and communicating design problems and solutions.

Later, Buschmann, Meunier, Rohnert, Sommerlad, and Stal (1996) introduced additional patterns and named three different types of patterns.

- Architectural Patterns — "Fundamental structural organization schemas for software systems. They provide a set of predefined subsystems, specify their responsibilities, and include rules and guidelines for organizing the relationship between them." (Buschmann, Meunier, Rohnert, Sommerlad, and Stal, 1996, p.25)
- Design Patterns — "A commonly-recurring structure of communicating components that solve a general design problem in a particular context." (Buschmann, Meunier, Rohnert, Sommerlad, and Stal, 1996, p.221)
- Idioms — "Low-level patterns specific to a programming language. An idiom describes how to implement particular aspects of components or relationships between them with the features of the given language." (Buschmann, Meunier, Rohnert, Sommerlad, and Stal, 1996, p.345)

This dissertation provides both architectural and design patterns and defines this group as *software patterns* as introduced by Schmidt, Fayad, and Johnson (1996).

Definition of Software Patterns — "A particular recurring design problem that arises in specific design contexts, and presents a well-proven generic scheme for its solution. The solution scheme is specified by describing its constituent components, their responsibilities and relationships, and the ways in which they collaborate." (Buschmann, Meunier, Rohnert, Sommerlad, and Stal, 1996, p. 8)

1.3.6 Variability

Originally, the concept of variability came from the automotive industry, in which different types of cars are manufactured based on a common set of parts (e.g. chassis, wheels or engine) (Thiel and Hein, 2002). This idea is later adopted in the domain of Software Product Lines (SPLs) (Pohl, Böckle, and Linden, 2005). Within SPL variability, the moment at which design decisions are taken, concerning the functionality supported by the software product, are delayed to later stages (Van Gurp, Bosch, and Svahnberg, 2001). The following binding moments for a variant are defined by Svahnberg, Gurp, and Bosch (2005):

- Product Architecture Derivation — Binding is done when a particular product architecture is generated.
- Compilation — The final product is bound during compilation, with specific variants according to compiler directives (e.g. different hardware platforms).
- Linking — Different components of the software product are linked just after compilation or before run-time. The bounding can be irrevocable or changeable every time the system runs.
- Run-time — Binding in this type of variability is, usually, defined in the code. New variants are configured while the product is running.

This dissertation focusses on *run-time* variability and does this in an online context.

Definition of Variability — "Variability in software architectures describes how well an architecture supports flexibility in a certain aspect, with an exact specification of the differences." (Galster and Avgeriou, 2011, p. 63)

1.4 Research Approach

In this section, the research questions posed in this dissertation, together with the research methods used, are discussed.

1.4.1 Research Questions

Software vendors offering Enterprise Software Applications are steadily moving from on-premises deployment models to SaaS deployment models. This shift enables them to share computing resources among their clients and benefit in terms of scalability and deployment costs. However, software vendors struggle to enable similar customization freedom as offered in their on-premises applications while keeping the advantages of the enabled economies of scale (D'souza, Kabbedijk, Seo, Jansen, and Brinkkemper, 2012). There is a need for variability in online enterprise software, but an overview of different options to implement variability is currently lacking.

The Main Research Question (MRQ) of this dissertation is stated as follows:

> **MRQ -** *How can variability in multi-tenant enterprise software be realized?*

In order to answer the main question of this dissertation, two Research Questions (RQs) are formulated. First we focus on gathering patterns related to variability in multi-tenant enterprise software, so that practical solutions in multi-tenant enterprise software engineering are identified. Second, the focus is more geared towards the architecting process, to put more emphasis on the role of software patterns in software development. Each RQ is answered in a different section of the dissertation and contains several additional sub-questions, which are discussed below, including the research methods used and, if applicable, a short explanation on how the results are validated.

RQ 1. *How can patterns be employed to implement variability in multi-tenant enterprise software?*
This question gives an overview of the main concepts relevant in online software deployment models and explores the role of software patterns in introducing variability. Related to the RQ, four sub questions are discussed:

RQ 1.1. *What is the concept of multi-tenancy?*
The term "Multi-Tenancy" is frequently used in academic literature and by practitioners, but to date, no comprehensive definition of Multi-Tenancy exists. Multi-Tenancy can be applied on different levels in the computing stack (i.e., database or instance level) and in different combinations (Mietzner, Unger, Titze, and Leymann, 2009), causing the

definitions to focus on different levels and combinations of levels. This RQ aims at giving an overview of the use of the term "Multi-Tenancy" by performing a systematic mapping study in academic and industrial literature. The results are validated by several cross validation checks during all steps in the mapping study process.

RQ 1.2. *How are software patterns used to implement variability?*
The implementation of variability in a software product is a challenge for software vendors. The problems regarding implementation of variability in online software can be assessed by using the structured approach patterns offer (Fowler, 2003). This RQ focusses on how the notion of software patterns can be used to document the problems, solutions and consequences related to implementing variability in Multi-Tenant SaaS products.

RQ 1.3. *What are the trade-offs of providing more variation in multi-tenant enterprise software?*
The effort needed to offer variability in online enterprise software greatly depends on the specific deployment model (Mietzner, Leymann, and Papazoglou, 2008) applied. The more resources, such as application instances or databases, are shared, the harder it can get to implement tenant specific variability. In this RQ the trade-off between variability and resource sharing is explored. Also, suggestions on offering variability in online enterprise software are provided in the form of different patterns. The patterns are gathered by performing two case studies and refined using a design science approach. The results are validated using expert interviews.

RQ 1.4. *How does the Command Query Responsibility Separation (CQRS) pattern influence the variability of a software product?*
A high level architectural pattern for designing scalable enterprise software is the CQRS pattern (Betts, Dominguez, Melnik, Simonazzi, and Subramanian, 2013). This pattern enables the distribution data and resources among different locations. Because of the encouraged distributed nature of the CQRS pattern, locations can be customized per tenant, leading to a high variability potential. This RQ examines the CQRS pattern and documents the different sub patterns and the influence of CQRS on the variability of online enterprise software. The patterns are gathered by performing a case study and validated using expert interviews and cooperative inquiry (Reason, 1994) during the case study.

RQ 2. *How can software patterns become an intrinsic part of the architecting process?*

Numerous solutions for implementing multi-tenancy or variability in multi-tenancy exist. This leads to a plethora of patterns and a struggle for software architects and decision makers to choose the best-fitting pattern. This question investigates the integration of software pattern decision-making in existing architecture processes and methods. The following sub questions are posed to answer the research question:

RQ 2.1. *How can software architects be supported in the selection process of choosing an applicable multi-tenant architecture pattern?*

Software vendors have different options to implement variability, each with different consequences. The options depend on the different levels in the computing stack on which resources are shared (Bhardwaj, Jain, and Jain, 2010). This RQ explores how software vendors can be supported in the decision process of choosing the appropriate pattern and focusses on structuring the different influences of the patterns on software quality attributes. The multi-tenant architecture patterns and pattern decision criteria are gathered by performing a structured literature study, combined with surveys among software architects to validate the results. The final patterns and their consequences are validated using an additional survey, conducted within three enterprise software companies.

RQ 2.2. *What are the influences of variability patterns on software quality attributes?*

Different ways of implementing variability in online software products exist. It is unclear, however, what the consequences are of specific solutions and what patterns are preferred in specific situations. This RQ addresses which different solutions (e.g., patterns) can be identified in current software systems. The 'variability patterns offer a problem statement, solution and consequences in terms of the effect on different quality attributes after implementation (e.g. maintainability or scalability) (ISO/IEC, 2011) and are gathered by performing three case studies. The results are evaluated and validated using expert interviews.

RQ 2.3. *How can software patterns be transparently evaluated and compared during the enterprise software architecting process?*

The last RQ is directed at proposing a method enabling the comparison of different patterns. First, the attributes on which the patterns will be

evaluated are identified, followed by the construction of a pattern evaluation method. The comparison of the patterns is based on quantitative assessment of the quality attributes affected by the implementation of the pattern. Expert focus groups are used to evaluate the patterns in a design science approach. The evaluation method is validated and adapted during the design science iterations.

By providing answers to all questions, an overview of how to implement variability in multi-tenant enterprise software is given. Additionally, software patterns will be constructed, which can be found in Appendix A.

1.4.2 Research Methods

In this dissertation, different research methods are used in order to gather all data and construct all artefacts to answer the research questions. An overview of all research methods and deliverables per chapter can be found in Table 1.2. This section discusses all research methods used.

Literature Study — Different types of literature study have been employed in this dissertation. Firstly, many studies perform a literature study, which follows no explicitly defined structure, with the aim of getting an exploratory view or vision of a specific topic. A more structured method to gather and analyse literature is the use of a Structured Literature Research (SLR) (Kitchenham, 2004). Within an SLR, all process steps in collecting and filtering the papers are documented (Kitchenham and Charters, 2007). Typically all papers are collected, resulting from a search string at a predefined number of academic digital libraries (Brereton, Kitchenham, Budgen, Turner, and Khalil, 2007), after which the initial collection of papers is filtered based on a set of inclusion and exclusion criteria (Fink, 2013). The resulting set is reduced further by filtering sequentially on different levels like title, abstract or full-text (Petticrew and Roberts, 2009). The goal of such a structured review is getting a complete overview of a research domain, based on all papers published on a specific topic.

When little papers exist on a topic, or the topic is too broad or scattered, an Systematic Mapping Study (SMS) is the appropriate method (Kitchenham, 2004). An SMS is used to map the field of a certain topic, instead of answering a specific research question (Petticrew and Roberts, 2009). The main difference with an SLR is that an SMS is more directed at uncovering research trends, instead of specific research questions (Kitchenham, Budgen, and Brereton, 2010).

Design Science — Research in the domain of Information Technology (IT) addresses design choices faced by practitioners (March and Smith, 1995). When a specific artifact (e.g., a software pattern) or method has to be developed, the appropriate application of design science is crucial for success (Peffers, Tuunanen, Rothenberger, and Chatterjee, 2007). Within design science, existing theories and models and literature, together with new data (for example from case studies), are used to develop new theories and artifacts. These results are later evaluated using *interviews*, *surveys* or *focus groups* (Tremblay, Hevner, and Berndt, 2010) and refined based on the evaluation results (Hevner, March, Park, and Ram, 2004).

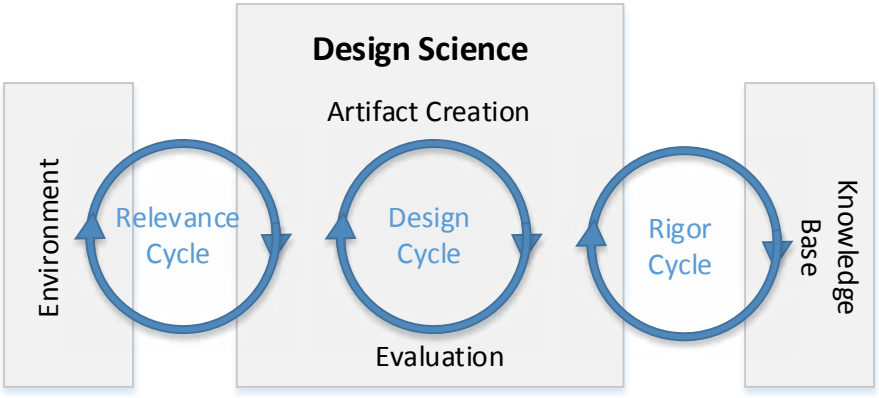
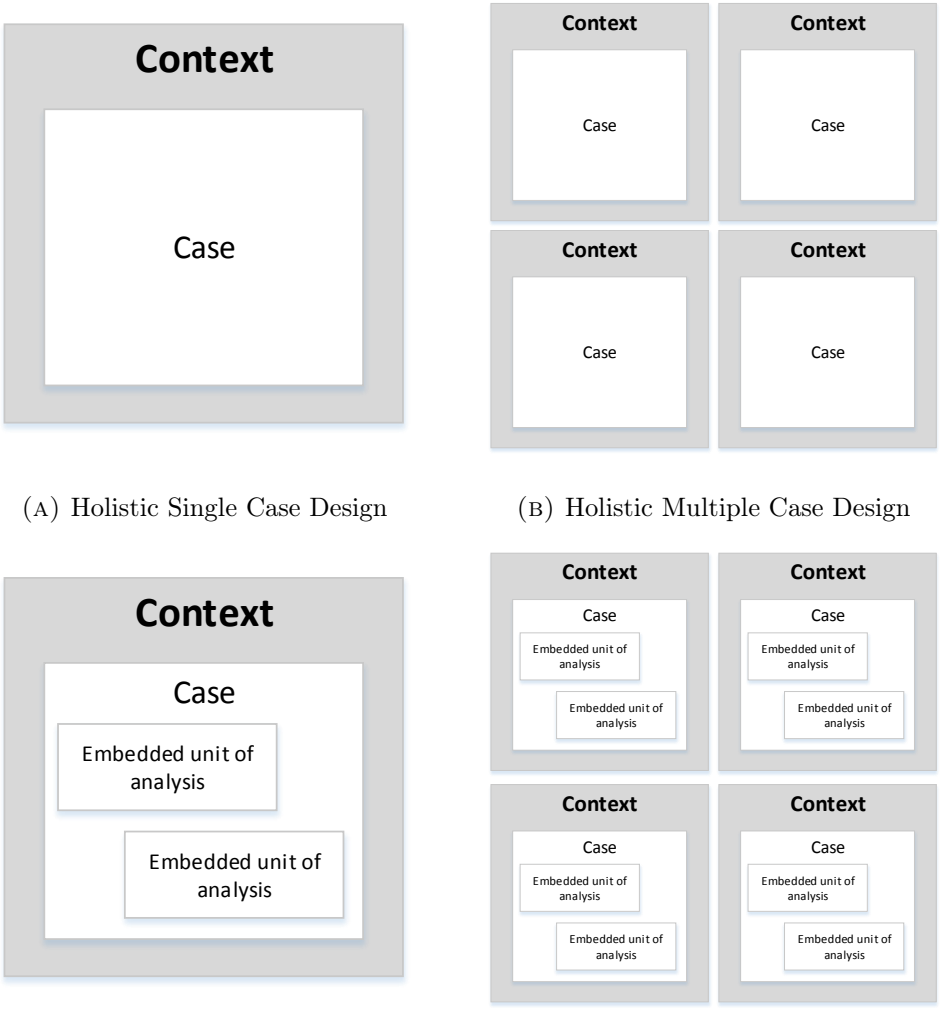


FIGURE 1.2: Design Science Research Method

The evaluation of the artifacts is a continuing process in design science, called the Design Cycle, as illustrated in Figure 1.2. During the design cycle there is a constant feedback loop with the real world (i.e. Environment) and knowledge questions (i.e. Knowledge Base) in respectively the Relevance and Rigor Cycle (Hevner and Chatterjee, 2010; Wieringa, 2009).

Case Study Research — Studying software architecture and architectural decision making, it is important to observe practices in real-world cases. Case study research is about the descriptive, exploratory or explanatory analysis of a person, event or organization (Eisenhardt, 1989). This dissertation focuses on the analysis of online enterprise software and the organizations and people related to it. Different types of case study design exist, as illustrated by Yin (2009).

The design shown in Figure 1.3a is a holistic single case design in which only one case is analysed. This can, for example, be the analysis of one software product in a single company. Figure 1.3b shows a similar design, but now multiple different cases are analyzed in order to gather data. Studying similar software products in different companies is an example of the holistic multiple case design.



(A) Holistic Single Case Design

(B) Holistic Multiple Case Design

(C) Embedded Single Case Design

(D) Embedded Multiple Case Design

FIGURE 1.3: Four types of case study designs

Figure 1.3c and 1.3d are representations of embedded case designs. In an embedded case design, several different units (e.g. software products or design problems) are analyzed in the same context. Identifying software patterns is ideally based on an embedded multiple case design; performing research at multiple companies, analyzing several design problems at each of them.

Case study research is sometimes criticised for being too specific and only geared towards generating hypothesis, leading to a lack of generalizability (Flyvbjerg, 2006). This critique can be mitigated by representative case selection. Furthermore, as the research presented in this thesis aims to develop theory, case studies enable the creation of high fidelity empirical research results that can support the creation of new theories and artifacts, that can later be validated by practitioners.

When appropriate cases are selected, and case study protocol is followed rigorously, case study research provides valuable results in the software engineering domain (Runeson and Höst, 2009).

1.4.3 Validation and Evaluation

When conducting research it is crucial to make sure the entire research process is setup and performed in a rigorous and valid way (Goodwin and Leech, 2003). The necessity of data validation and research method validity is also true in a research domain in which case study research plays a dominant role, e.g. software pattern research (Eisenhardt, 1989; Yin, 2009). By following prescribed guidelines for conducting research, such as recording interviews and using multiple sources of information (Runeson and Höst, 2009), the validity of the research process is warranted. Different types of validity (i.e. construct, internal and external validity) are discussed throughout the dissertation. Many artefacts (e.g. patterns) are, additionally, evaluated in order to assess or measure their properties, based on certain standards. The following definitions are used in this dissertation:

- Validation — To check whether the results are corroborated on a sound or authoritative basis (Merriam Webster Online, 2014b).
- Validity — To ensure measurements are well-founded and correspond to the real world (Goodwin and Leech, 2003).
  - Construct Validity — The degree to which a test measures what it claims, or purports, to be measuring (Cronbach and Meehl, 1955).
  - Internal Validity — The extent to which a causal conclusion based on a study is warranted (Shadish, Cook, and Campbell, 2002).
  - External Validity — The extent to which the results of a study can be generalized to other situations and to other people (Mitchell and Jolley, 2012).
- Evaluation — The assessment of the value or condition of a certain concept or attribute (Merriam Webster Online, 2014a).

Within this dissertation, all results from data gathering and analysis are validated to make sure the research was conducted carefully and diligently. Table 1.1 shows the different measures used within this dissertation to validate the results.

Validation Measure	Chapter
Cross validation among authors	2
Interviews	4, 5, 6, 7
Focus Groups	4
Cooperative Inquiry	5
Surveys	6
Pilot Sessions	8

TABLE 1.1: Validation measures used per chapter

The validation measures mentioned in Table 1.1 are employed in this research in various ways. When *cross validation among authors* is performed in this dissertation, data collected by, or conclusions drawn by, one researcher are compared to data or conclusions brought up by another researcher. If differences exist between the two, the results are discussed and adjusted accordingly. During *interviews*, experts are asked about their experiences and vision on certain topics. The interview can consist of a list of questions (i.e. structured interview) or can be more open for discussion (i.e. semi-structured interview) (Berg and Lune, 2004). Surveys, in this research, are used in a similar way to structured interviews, but the questions and answers are communicated on paper. In *focus groups*, a group of experts is interviewed simultaneously, often in a semi-stuctured nature (Kitzinger, 1995). In cooperative inquiry, during data gathering and analysis, the researcher takes a place within the case company. Research results can be immediately and constantly validated by discussing them with the experts in the case company (Reason, 1994). Finally, *pilot sessions* are applied within this research, in which a candidate method is tested by trying it out in a controlled environment with potential future users.

Each chapter in this dissertation will discuss these concepts, if applicable, in more depth in the validation section.

1.5 Dissertation Outline

Chapters 2 to 8 of this dissertation each match exactly one sub-question, as presented in Section1.4.1. An overview of the questions, methods and contributions per chapter can be found in Table 1.2. The outline of the dissertation is as follows:

Chapter 1— *Introduction*

This chapter explains the relevance of this research, from both an academic

Ch.	RQ	Research Method	Contributions
2	1.1	Literature Study (SMS)	• Multi-tenancy Definition • MT Research Agenda
3	1.2		• Pattern Vision Statement
4	1.3	Design Science Case Study Research	• Customizable Data Views Pattern • Module Dependent Menu Pattern • Pre/Post Update Hooks Pattern
5	1.4	Design Science Case Study Research	• Command Query Responsibility Separation (CQRS) Pattern • 7 CQRS sub Patterns
6	2.1	Literature Study (SLR) Design Science	• Multi-tenant Architecture Assessment Model (MAAM) • 12 Multi-Tenant Architecture (MTA) Patterns • MTA Decision Matrix • MTA Assessment Rules of Thumb
7	2.2	Case Study Research	• Component Interceptor Pattern • Event Distribution Pattern • Datasource Router Pattern • Custom Property Object Pattern
8	2.3	Design Science	• Software Pattern Evaluation Method (SPEM)

Table 1.2: Overview of RQs, research methods and contributions per chapter

and industrial perspective. Besides a discussion on important concepts related to this research, an overview is given of all research questions. The research methods used in the different chapters are explained as well. Also, an outline of the dissertation can be found in this chapter.

Chapter 2— *Defining Multi-Tenancy*

The concept of multi-tenancy is discussed in this chapter. Based on an SMS of academic papers and industrial blogs, a definition of Multi-Tenant (MT) is constructed, together with a research agenda aimed at structuring future multi-tenancy research. This chapter is accepted as a full research paper in the Journal of Systems and Software (JSS) (Kabbedijk, Bezemer, Jansen, and Zaidman, 2014 (In Press)).

Chapter 3— *The Role of Variability Patterns*

In this chapter a conceptual model is presented, illustrating the role software

patterns play in solving variability problems. This chapter has been published as an invited paper at the International Workshop on Variability in Software Architecture (VARSA) (Kabbedijk and Jansen, 2012).

Chapter 4— *Variability in Multi-tenant Systems*

This chapter introduces three variability patterns (i.e. Customizable Data Views, Module Dependent Menu and Pre/Post Update Hooks) which enable variability in online software applications. Also, the trade-off between variability and deployment model is discussed. This chapter has been published as a full research paper at the International Conference on Conceptual Modeling (ER) (Kabbedijk and Jansen, 2011).

Chapter 5— *Variability Consequences of the CQRS Pattern*

The relationship between the CQRS pattern and variability is discussed in this chapter. Together with the CQRS pattern, seven sub patterns are discussed which can be implemented to support CQRS. This chapter has been published as a full paper at the European Conference on Pattern Languages of Programs (EuroPLoP) (Kabbedijk, Jansen, and Brinkkemper, 2012).

Chapter 6— *Multi-Tenant Architecture Assessment*

In this chapter, Multi-tenant Architecture Assessment Model (MAAM) is presented, which helps in the selection of the most fitting Multi-Tenant Architecture (MTA). Twelve patterns are provided, together with a comparison of the patterns and five rules of thumb. The rules of thumb can be used as initial guidelines in the architecting process. A shortened version of this chapter has been published at the European Conference on Software Architecture (ECSA) (Kabbedijk, Pors, Jansen, and Brinkkemper, 2014).

Chapter 7— *Comparing Dynamical Adaptation Patterns*

This chapter discusses two problems in variable multi-tenant enterprise software and presents two patterns for each of the problems. A qualitative comparison between the patterns is provided. A shorter version of this chapter has been published as a full research paper at the International Conference on Pervasive Patterns and Applications (PATTERNS) (Kabbedijk, Salfischberger, and Jansen, 2013). The extended chapter has been published as a full research paper at the International Journal on Advances in Software (Kabbedijk, Jansen, and Salfischberger, 2014).

Chapter 8— *Software Pattern Evaluation Method*

In this chapter, a Software Pattern Evaluation Method (SPEM) is presented.

The method prescribes the evaluation of software patterns and aims at providing software architects with a way to assess the effect of applying patterns, on software quality. This chapter has been published as a full research paper at the International Conference on Pervasive Patterns and Applications (PATTERNS) (Kabbedijk, Donselaar, and Jansen, 2014).

Chapter 9— *Conclusion*

This chapter gives an overview of the answers to all research questions. Also, a discussion of the contributions, implications and future work is provided.

For easy reference, a pattern catalogue is included in Appendix A, providing an overview of all software patterns originating from this dissertation.

Part I

Variability and Multi-tenancy

Chapter 2

Defining Multi-Tenancy

Abstract

Software as a service is frequently offered in a multi-tenant style, where customers of the application and their end-users share resources such as software and hardware among all users, without necessarily sharing data. It is surprising that, with such a popular paradigm, little agreement exists with regard to the definition, domain, and challenges of multi-tenancy. This absence is detrimental to the research community and the industry, as it hampers progress in the domain of multi-tenancy and enables organizations and academics to wield their own definitions to further their commercial or research agendas.

In this article, a systematic mapping study on multi-tenancy is described in which 761 academic papers and 371 industrial blogs are analysed. Both the industrial and academic perspective are assessed, in order to get a complete overview. The definition and topic maps provide a comprehensive overview of the domain, while the research agenda, listing seven important domains, provides a roadmap for future research efforts.

This work has been accepted as *Defining Multi-Tenancy: A Structured Mapping Study on the Academic and the Industrial Perspective* (Kabbedijk, Bezemer, Jansen, and Zaidman, 2014 (In Press)). It is co-authored by Cor-Paul Bezemer, Slinger Jansen and Andy Zaidman.

2.1 Introduction

An ongoing growing influence of cloud computing and Software-as-a-Service (SaaS) can be observed in the enterprise software domain (Forbes, 2014). One of the key features of SaaS is the ability to share computing resources in offering a software product to different customers. To benefit from this ability, the architecture of SaaS products should cater for the sharing of software instances and databases. A popular architectural style for achieving this is known as Multi-Tenancy. The concept of multi-tenancy, within the software architecture community, is usually referred to as the ability to serve multiple client organizations through one instance of a software product and can be seen as a high level architectural pattern in which a single instance of a software product is hosted on the software vendor's infrastructure, and multiple customers access the same instance (Bezemer, Zaidman, Platzbeecker, Hurkmans, and Hart, 2010). The specific method for sharing instances (e.g. reentrancy or queueing) is generally not specified within the multi-tenancy pattern. Multi-tenancy allows for the customization of the single software instance according to the varying requirements of many customers (Kwok, Nguyen, and Lam, 2008), contrasting with the multi-user model in which there is no substantial variability (Bezemer and Zaidman, 2010). Also, multi-tenancy is one of the key factors for achieving higher profit margins by leveraging the economies of scale (Guo, Sun, Huang, Wang, and Gao, 2007).

Multi-tenancy has evolved from a number of previous paradigms in information technology. More concretely, starting in the 1960s companies performed *time-sharing*, they rented space and processing power on mainframe computers to reduce computing expenses; often they also reused existing applications (Wilkes, 1975). Around 1990 the *application service provider* (ASP) model was introduced, where ASPs hosted applications on behalf of their customers. ASPs were typically forced to host applications on separate machines or as separate processes (Smith and Kumar, 2004). Finally, the multi-user model is most-known from popular consumer-oriented web applications (e.g. Facebook) that are functionally designed as a single application instance that serves all customers (Bezemer and Zaidman, 2010). Multi-tenant applications represent a natural evolution from these previous paradigms. Similarly, around the year 2000, Bennett, Layzell, Budgen, Brereton, Macaulay, and Munro, 2000 set out a vision for service-based software applications, in which they note a number of essential ingredients for what we now call multi-tenancy, namely: demand-led provisioning of software services and a high degree of personalization of software.

In the domain of software (and hardware) systems, the topic of multi-tenancy in scientific literature appeared relatively recently, with the first explicit mention of the term in a paper by Chong and Carraro, 2006 in the MSDN Library. Within multi-tenancy, the hardware and software infrastructure is shared and a hosted application can serve user requests from multiple companies concurrently (Guo, Sun, Huang, Wang, and Gao, 2007). Multi-tenancy is regarded a key attribute of well-designed SaaS applications by Chong and Carraro, who developed a commonly used maturity model of SaaS that distinguishes four maturity levels. The last two maturity levels in this model describe multi-tenancy, rendering it as a requirement for a mature SaaS application. Multi-tenancy is not confined to specific resources, but is applicable at different levels in a system's architecture, for example on a database or instance level. As a result, various approaches to a multi-tenant architecture are possible (Osipov, Goldszmidt, Taylor, and Poddar, 2009; Natis, 2008).

Most academics and practitioners agree multi-tenancy enables software vendors to serve multiple customers from a single online product, but specific implementations differ significantly, leading to an indistinct understanding of the different levels to which multi-tenancy can be applied. This varying definition of multi-tenancy is confusing *among* academics and practitioners, but it also complicates the communication *between* them, caused by the different understanding of multi-tenancy among them. Oracle, for example, looks at multi-tenancy primarily from a database perspective (Oracle, 2009), while Microsoft looks at multi-tenancy more from a functional perspective (Microsoft, 2012).

The goal of this paper is to chart and bridge these varying definitions and the views from both industry and academics on multi-tenancy. First, there is a need for an overview of the different definitions of multi-tenancy, followed by a clear analysis of what is shared among the different definitions. Having such an overview will improve the understandability of multi-tenancy and allows parties to be more aware of the varying nature of the definitions on multi-tenancy at this moment. Establishing common ground also allows us to define research challenges to guide future research in the domain of multi-tenancy. This paper aims at satisfying these needs by performing a structural search in academic literature and blog posts, as described in Section 2.2. All search data is analysed (Section 2.3) and an overview of the results can be found in Section 2.4. The different perspectives on multi-tenancy emerging from the results are synthesized to one overarching definition (Section 2.5). To structure future research, a research agenda containing seven

areas of interest is proposed (Section 2.6), followed by a conclusion and discussion in Section 2.8.

2.2 Research Method

In order to get an overview of the current state of multi-tenancy literature and get insight on the interpretation of multi-tenancy from different perspectives a set of research questions has been constructed. The main research question (RQ) is as follows:

RQ: *How to characterize multi-tenancy?*

The main research question is addressed by answering the sub research questions (SubRQs) listed below. Each question focusses on a different perspective on the characterization of multi-tenancy.

SubRQ1: *What comprehensive definition for multi-tenancy can be constructed based on current literature?*
Rationale: Multi-tenancy is not a new concept, and many different definitions already exist. Since these definitions may reflect different perspectives on a software product and focus on different elements, an overall definition should be developed.

SubRQ2: *How is multi-tenancy interpreted in academia and industry?*
Rationale: The use or understanding of the concept of multi-tenancy in industry could differ from the common use in academia. This possible chasm between academia and industry inhibits cooperation and communication between both domains. To examine this, not only academic papers are analyzed, but also 300 internet blog results are used to be able to compare uses in both domains.

SubRQ3: *What future research topics can be defined based on current literature?*
Rationale: Since the domain of multi-tenancy research is rather young and scattered, there is a need for guidance on future research. Several research topics are distilled from the academic literature.

The questions are answered based on the academic papers and public blogs aggregated by the systematic search and selection process that is followed in this

research. Two different datasets are gathered and analyzed using a Systematic Mapping Study (SMS) approach. The first dataset is gathered from within the academic domain, while the second dataset is composed from blogs from the industry domain. An SMS is the appropriate method when trying to answer a general research question on a certain topic (Kitchenham, Budgen, and Brereton, 2010) and provides a detailed overview of the topic. A previous paper by Anjum and Budgen, 2012 was used as a guideline for reporting the mapping study.

2.2.1 Academic Literature Collection

In order to identify, evaluate and interpret the available literature relevant to a particular topic in an unbiased, objective and systematic way, common practice is to perform a Systematic Literature Review (SLR) (Budgen, Turner, Brereton, and Kitchenham, 2008). The proper execution of an SLR is still something that is not done frequently in the field of Software Engineering (SE) (Kitchenham, Pearl Brereton, Budgen, Turner, Bailey, and Linkman, 2009). This is probably caused by the fact that an SLR is time-consuming and should be performed rigorously within a mature research domain. However, if little evidence exists or the topic is too broad or scattered, then a Systematic Mapping Study (SMS) is the appropriate method (Kitchenham, 2004). An SMS is used to map the field of a certain topic, instead of answering a specific research question (Petticrew and Roberts, 2009). Since the research domain of multi-tenancy is not mature yet and initial search shows definitions differ significantly, this study uses an SMS to get a overview of the concept of multi-tenancy. This paper presents an SMS in which the different perspectives on multi-tenancy are examined.

The systematic mapping study was performed according to the phases described by Peterson et al. (Petersen, Feldt, Mujtaba, and Mattsson, 2008). First, a search for relevant publications was performed, second a classification scheme was constructed, and third, the publications were mapped. The details of the different steps are described below. The first phase consisted of literature retrieval. The steps and the resulting dataset size are as follows:

1. *Search Execution* — Dataset retrieval from using the search query on the following databases: ACM, CiteSeerX, IEEE, ISI, Science Direct, Scopus, SpringerLink, and Wiley. Since Google Scholar aggregates from all the databases listed, it was excluded from the search to minimize the number

of duplicates. The search has been performed using the following keyword query:

> "multi-tenancy" OR "multi-tenant" OR multitenancy OR multi-tenant OR "multi tenancy" OR "multi tenant"

2. *Paper Screening* consists of a check for completeness, relevance, and compliance to the inclusion and exclusion criteria. Included papers are peer reviewed academic papers. Excluded are non-English papers and duplicates not identified in the previous step.

3. *Filtering on Title and Year* — Deletion of papers written before 2000 because the term multi-tenancy in this field was non-existent before that year. Papers describing multi-tenancy unrelated to IT (e.g. related to housing) are excluded.

4. *Filtering on Abstracts* — Papers that merely use the term but do not actively discuss multi-tenancy are removed as well.

5. *Filtering on Full Text* — The final selection was based on the criteria that the paper must either explicitly state a multi-tenancy definition or refer to one instead.

The results of conducting all five steps were systematically logged in a central database accessible by all authors. After each step, 10% of all papers have been selected by querying every 10^{th} entry in the database, and checked for inter-rater agreement by all authors. If a paper was rated differently by another author, the discrepancy was discussed and corrected. When more than one discrepancy was identified, the step was redone. This inter-rater agreement check was done in order to ensure construct validity of the data gathering (Eisenhardt, 1989).

2.2.2 Industrial Literature Collection

The gathering of industrial literature (i.e. blogs), was performed in order to provide a sanity check for the academic literature. The results were not used explicitly for the construction of the multi-tenancy definition or research agenda, but serve

to examine potential different interpretations of multi-tenancy between industry and academia. For the industrial perspective of this survey, we have mirrored the process of the Systematic Mapping Study for scientific literature. We use the same three phases that Petersen, Feldt, Mujtaba, and Mattsson, 2008 describe for the traditional SMS, being:

1. *Search Execution* — Consists of dataset retrieval from using the search query. We use the same search query as for the scientific literature, but this time applied it to the traditional Google search and the Google Blog search (`www.google.com/blogsearch`). The search string used was:

   > "multi-tenancy" OR "multi-tenant" OR multitenancy OR multi-tenant OR "multi tenancy" OR "multi tenant"

   The search results are limited to the first 300 results of the traditional Google Search and to 100 of the Google Blog search. This cut-off is instigated to keep the results manageable, but we also found that around these thresholds the search results become decreasingly relevant (e.g., the traditional Google search started returning results that were not-related to multi-tenancy in the area of computer science).

2. *Website Categorization* — The first 100 entries of the traditional Google search are screened and subsequently the second and fourth authors of the paper established an initial categorization of the web sites that were encountered. The categorization is first performed by both authors independently, after which the initial sets are compared and discussed. Based on discussion, the final set is constructed. Having a website categorization, makes it easier to understand the importance of multi-tenancy in industry and how we could learn from these web sites when considering how multi-tenancy is defined and used in industry.

3. *Inter-rater agreement* — The categorization of the websites is done by the second and fourth author. Both of them categorize half of the website entries. In order to achieve inter-rater agreement 10 websites entries from the second author and another 10 from the fourth author were exchanged and re-classified by the other.

4. *Investigation of Full Text* — Because a web site typically does not have the same structure as a scientific paper, we screened the full text of each web site in full in order to determine (1) whether the search result is within the

scope of this study and (2) in which category the website should be placed. The scope was determined to be everything related to IT.

Whenever differences existed in the classification done by the second and fourth author, an agreement is reached through discussion. The classification result and similar classifications are adjusted according to the new joint interpretation.

2.3 Classification

2.3.1 Academic Literature Classification

In this section, the analysis of the academic literature is illustrated. An overview of the results per phase in the systematic mapping study is presented below, followed by a top-down approach for the literature analysis.

1. *Search Execution* — The search resulted in 1371 papers. After duplicate removal based on title, a database of 761 papers was created.
2. *Paper Screening* — This phase resulted in 672 applicable papers.
3. *Filtering on Title and Year* — Resulted in 259 applicable papers.
4. *Filtering on Abstracts* — After filtering, 92 applicable papers were identified.
5. *Filtering on Full Text* — This resulted in 48 applicable papers.

After checking for the inter-rater agreement in each step, small discrepancies between the raters were found. None of the steps, however, had a discrepancy larger than one paper, which meant none of the steps had to be redone. The small level of discrepancy can be explained by the fact both authors are knowledgeable in the area of multi-tenancy and already knew many of the papers published within this domain.

Different publication types are discussed in Figure 2.1, showing an overview of the different paper publication outlet types. Conferences clearly play a dominant role in publishing papers on multi-tenancy (27 papers), followed by journals (18 papers). Only three papers were found in workshops.

To further investigate the state of the art in the scientific literature an analysis on the research was performed as well as classification by research type. This

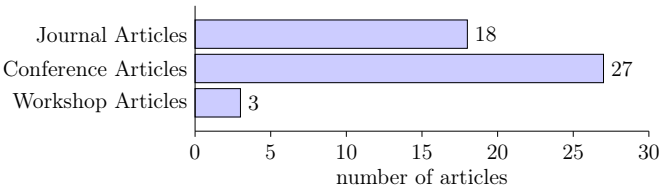


FIGURE 2.1: Publication outlets for academic articles on multi-tenancy

overview is useful for identifying gaps in current literature. To classify the type of research approach, six existing distinct research categories were used (Wieringa, Heerkens, and Regnell, 2009). An overview of these type of research approaches is presented in Table 2.1.

Category	**Description**	**N**
Solution Proposal	Proposes a solution with arguments for its relevance without an evaluation in practice but a proof-of-concept is acceptable.	26
Validation Research	Investigates an existing solution and validates it by using a sound scientific approach.	10
Evaluation Research	Investigation of a problem or implementation of a technique in practice.	6
Philosophical Paper	Introduces a new view on a subject, a new concept, conceptual framework.	5
Experience Paper	Explains why or how something has been done in practice. For example lessons learned from projects.	1
Opinion Paper	Contains an author's opinion on a subject.	0

TABLE 2.1: Categorization of 48 papers, listing the number of occurrences (N) for each type of paper encountered.

Papers were classified using an evolutionary approach, where subjects are selected based on title, abstract and keywords. Papers are categorized and categories are evolved throughout the review using splitting and merging. The analysis of the results focuses on presenting the frequencies of publications for different research categories. An overview of popular and less popular categories can be used to identify gaps and possibilities for future research. It also provides a picture about the nature of the scientific material and the maturity of the field. The results from this analysis are depicted in Table 2.2. Please note the last research category (i.e. Opinion Paper) is not included in the table, since no papers were part of this category.

The list of topics is based on the abstracts of the papers and the keywords listed. It is possible one paper discusses multiple topics, in which case it is listed on all of these topics. A paper, however, is always part of only one research category.

	Evaluation Research	Solution Proposal	Validation Research	Philosophical Paper	Experience Paper	Total
SaaS	4	19	6	2	1	32
Architecture	4	13	7	3	1	28
Implementation	2	8	2	2	1	15
Database	-	4	6	2	1	13
Balancing & Placement	2	6	2	3	-	13
Variability	1	8	1	-	1	11
Infrastructure	1	5	3	1	-	10
Industry Evaluation	1	4	1	2	1	9
Quality Assurance	1	6	1	-	-	8
Platform Development	-	4	2	1	-	7
Security	-	3	1	2	-	6
Standards	-	3	-	2	-	5
Total	16	83	32	20	6	

TABLE 2.2: Multi-tenancy research topics per research category

2.3.2 Industrial Literature Classification

This section presents the results of the industry literature gathering per phase, followed by a discussion of the analysis.

1. *Search Execution* — Among the results were a number of scientific papers, all of which were also part of our search for scientific literature. After removing duplicates, this resulted in 371 entries.

2. *Website Categorization* — Eight categories were identified, as shown in Table 2.3. The first half of the websites was categorized by the second author, the second half was categorized by the fourth author.

3. *Inter-rater agreement* — To validate the choice of categories and evaluate the categorization process, a random sample (N=12) of websites was categorized by both the second and fourth author and compared afterwards. Small changes existed in the classification, mainly due to different interpretation of the categories. In 75% (9/12) of the cases, both authors completely agreed on the categorization (average of 2.33 categories per website). In the three other cases, they at least partly agreed on the categorization. Considering a website can be categorized in a subset of unknown size of 8 different categories, we considered this to be a good level of inter-rater agreement.

4. *Investigation of Full Text* — All of the 371 entries appeared to be relevant to the concept of multi-tenancy in IT.

As mentioned in Section 2.2.2, we started out by analyzing the first 100 entries returned by Google to create an initial categorization of search results. Small changes to the categorization were made while analyzing all search entries. The final categories that we ended up with are listed in Table 2.3.

Table 2.3 also describes the criteria that we used for the categorization process. Note that we tried to distinguish "corporate opinions" from "individual opinions" as much as possible, hence the many different categories. From the initial search results we removed duplicates, and excluded 14 academic papers and dead website links. This resulted in 371 search entries being investigated, divided over the aforementioned categories. An overview can be seen in Table 2.3. It should be noted that some search results were categorized in multiple categories, for example, a corporate blog might also contain an explicit advertisement for the product being described.

Category	**Description**	**N**
Non-corporate blog	A software engineer or technology expert writing about multi-tenancy. No (corporate) affiliation is mentioned or could be retrieved.	117
Corporate blogs	White papers mentioning multi-tenancy. This category consists of web pages that are either hosted by a corporation or that explicitly state that the author or text was written from a specific company's perspective. It does not directly advertise the services of the company with regard to multi-tenant technologies, but it describes the company's vision on multi-tenancy.	84
Howto	Web page describing how to implement multi-tenancy. No corporate affiliation or link to a specific product is mentioned.	82
Advertisement	Web page advertising a product or service related to multi-tenancy.	81
Evangelism	Web page containing a strong opinion either in favor or against multi-tenancy	79
Definition	Web page containing a definition (or a discussion on the definition) of multi-tenancy	38
Support forum	Forum discussing multi-tenancy. This forum can be product-specific or product-agnostic. Some support forums are hosted by corporations, others are hosted by StackOverflow, Google Groups, etc.	36
Product manual	Web page describing how to use a multi-tenancy oriented product or service. This category of websites can be linked to a specific product or service.	18

TABLE 2.3: Categorization of 371 Google search entries, listing the number of occurences (N)

2.4 Observations

This section presents a set of observations, based on the results of the Academic and Industrial result classification. All observations were discussed among all four authors and adapted if needed. The observations do not aim to provide a complete list, but rather give a representative illustration of the multi-tenancy domain.

2.4.1 Academic Paper Results

Based on the paper classification in Section 2.3.1, the following observations are made:

Observation 1: Conference oriented — As Figure 2.1 shows, around 56% of all research papers on multi-tenancy are published in conference proceedings, compared to 37.5% in journal publications and only around 6.5% in workshop proceedings. The accent on conference publications in not uncommon in the IT domain, but the lack of workshop publications is striking. One such distribution could indicate a very mature research domain, but considering the novelty of multi-tenancy and number of papers published this is unlikely. A more plausible cause is that the domain of multi-tenancy research has no strong community yet and workshops still have to be formed, causing researchers to submit results to conferences and journals, which often have a broader scope.

Observation 2: Many proposals, lack of experience — Table 2.2 shows a strong emphasis on solution proposals and only one paper reporting on industrial experiences. This imbalance indicates that the research domain is still not mature, and that most of the solutions proposed have not yet been implemented or evaluated. The large difference can also signal the lack of cooperation between industry and academia.

Observation 3: Architecture and SaaS play a big role — Unsurprisingly, the topics of SaaS (32 papers) and architecture (28 papers) are addressed a lot in multi-tenancy research. Multi-tenancy is clearly positioned as an architectural tactic for online software. Since SaaS and architecture refer to the entire software stack, this observation also shows that research focusses on the complete software product instead of just one level (e.g. Database).

2.4.2 Blog Post Results

We did a full reading of three categories of web pages, being web pages or blog posts in the categories *non-corporate blog*, *corporate blog*, *definition* and *evangelism*. This reading gave us an impression of some of the advantages, disadvantages and/or issues that practitioners see or have with multi-tenancy. We have translated the impression that we thus got into the following observations:

Observation 1: Different multi-tenancy levels — Some practitioners make a distinction between multi-tenancy at the level of the *infrastructure* (multiple operating system instances on the same physical hardware), at the level of the *platform* (different applications and/or tenants on the same instance of the operation system) and at the *application* level (a single run-time stack is shared with multiple tenants). While not every blog post or website is perfectly clear on this, we observe that most websites on multi-tenancy are actually about the infrastructural or platform level application of multi-tenancy.

Observation 2: Cloud-based nature — For many practitioners multi-tenancy is *evident* in a cloud-based setting (IBM, 2011). This points at two distinct issues with how multi-tenancy is perceived by practitioners. First, a cloud environment is — by its very purpose — a shared platform environment, which in turn indicates that multi-tenancy is seen by many as another way of saying *Platform as a Service* or PaaS. Indeed, in a PaaS setting, tenants can rent a piece of shared platform which can consist of an operating system and standard server applications like a web server, a database, etc. Secondly, in some cases, practitioners were also considering multi-tenancy at the level of software in a cloud-based setting. In this context, practitioners were considering that Software as a Service offerings can be offered more efficiently if the underlying platform is elastic.

Observation 3: Configurability of multi-tenant applications — Configurability, or variability, of multi-tenant applications is seldomly mentioned. This raises two interesting points:

- As discussed in Observation 1 this may hint at a greater awareness of multi-tenancy at the infrastructural or platform level, where configurability might not be so much of an issue
- There is no apparent need for the configurability of multi-tenant software applications, which might indicate that most applications are actually *multi-user* applications or applications that share resources but that do not offer (advanced) forms of configurability.

When customization is discussed, it is clear that customization should lead to a tailored experience for each tenant and that customization should be done by configuring application metadata. As such, configurability requires no programming. Another important point mentioned is that customizations for one client should not affect other clients.

Observation 4: Multi-tenant database — A number of websites explicitly mention the database as being multi-tenant. In this situation different applications share a single database. When a single multi-tenant application is using the database, some web site authors express concern about data separation, i.e., making sure that tenants do not get access to another tenant's data.

2.5 Definition

A total of 43 different definitions was extracted from the academic literature with the aim of finding the best definition for use in the multi-tenancy domain, that describes the relevant elements, but also at all levels at which multi-tenancy is possible.

Word	Occurrence
Instance	26
Application	24
SaaS	22
Multiple	21
Infrastructure	20
Single	18
Software	15
Customer	15
Share	13
Database	12

TABLE 2.4: Word Frequency Analysis

Identification: The 43 definitions were identified by manually searching through papers for terms such as "we define multi-tenancy" or "multi-tenancy is defined as". A common observation from these definitions is that these are typically poorly formulated and only applicable at one level of the software stack or infrastructure. An example: "A multi-tenant cloud system allows multiple users to share a common physical computing infrastructure in a cost-effective way" (Du, Gu, and Reeves, 2010). This definition is not generic, but refers specifically to a "system". Its strong points are the "common physical computing infrastructure" and its emphasis on "costs", one of the main drivers of multi-tenancy. Another definition is "Multi-tenancy allows a single application to emulate multiple application instances" (Azeez, Perera, Gamage, Linton, Siriwardana, Leelaratne, Weerawarana, and Fremantle, 2010). This definition speaks specifically of an application, thereby excluding for instance hardware resources or databases.

Word Frequency Analysis - An analysis of frequent occurrences of terms was performed to find the main concepts in multi-tenancy definitions. The results of this analysis can be found in Table 2.4. Obviously, relevant aspects of multi-tenancy are the fact that something (single) is being shared among multiple customers, that it takes place on several levels (system, service, application, database, and infrastructure), and that it changes traditional modes of service or software delivery. To clarify, we have conceptualized a system, such that we can reuse it for the definition later in Figure 2.2. The dotted boxes are parts of the system that are not influenced by software level multi-tenancy. Efforts exist to apply multi-tenancy at the middle-ware level (Strauch, Andrikopoulos, Gómez Sáez, and Leymann, 2013), but we did not explicitly analyse this, for the sake of creating a high level general definition.

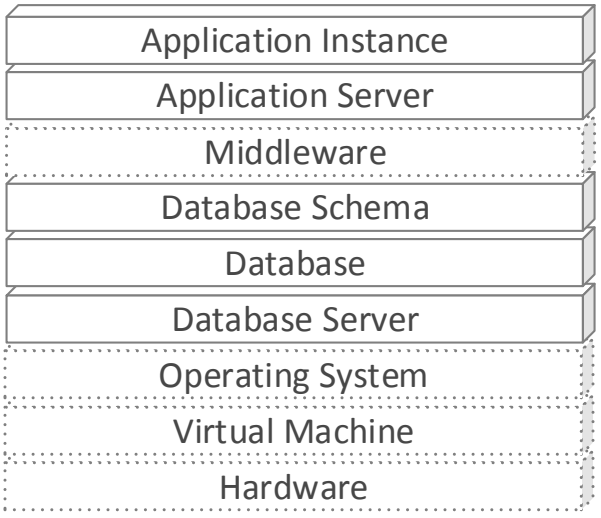


FIGURE 2.2: Software Stack: The different system levels where multi-tenancy can be applied to share resources.

Checklist: A checklist containing five criteria was constructed for use in this research, in order to assess the quality of all definitions. The list is based on five principles discussed by Copi and Miller (1972). Furthermore, for each definition we attempted to establish whether it was abstract enough to play a part on all three levels (service, database, and infrastructure). The criteria were formulated as follows:

- A definition must set out the essential attributes of the thing defined.
- Definitions should avoid circularity.

- The definition must not be too wide or too narrow. It must be applicable to everything to which the defined term applies (i.e. not miss anything out), and to nothing else (i.e. not include any things to which the defined term would not truly apply).
- The definition must not be obscure.
- A definition should not be negative where it can be positive.

Several definitions were selected to establish a baseline for the multi-tenancy definition in this paper, based on the criteria mentioned above. First, the definition given by Rimal, Choi, and Lump is "multi-tenancy is when common resources and a single instance of both the object code of an application and the underlying database are used to support multiple customers simultaneously" (Rimal, Choi, and Lumb, 2009). The definition includes relevant aspects of multi-tenancy, such as "multiple customers" and "common resources" and it speaks of all three levels on which multi-tenancy can play a part (database, service, and hardware resources). However, the definition lacks a goal statement (what is the advantage of multi-tenancy?). Another definition is given by Guo et al.: "In a multi-tenant enabled service environment, user requests from different organizations and companies (tenants) are served concurrently by one or more hosted application instances based on the shared hardware and software infrastructure." (Guo, Sun, Huang, Wang, and Gao, 2007). This definition too addresses only two levels, but adds multiple instances of the software. Finally, an interesting definition is "Multi-tenancy aims to enable a service environment that user requests from different tenants are served concurrently by the least number of hosted service instances running on the shared hardware and software infrastructure" (Li, Liu, Li, and Chen, 2008) which focuses on reducing costs by sharing resources. Based on the definitions stated above we define multi-tenancy as follows:

Definition: *Multi-tenancy is a property of a system where multiple customers, so-called tenants, transparently share the system's resources, such as services, applications, databases, or hardware, with the aim of lowering costs, while still being able to exclusively configure the system to the needs of the tenant.*

This definition caters to different needs. To begin with it mentions the most common terms used to identify multi-tenancy (with the sole exception of "instance",

but more on that later). Furthermore, it embraces any kind of system and its layers, from a complete service system with multiple instances (like Salesforce.com), to a simple hard drive that is shared among different end-users. Thirdly, it provides the main aim for applying multi-tenancy in a context, being the reduction of costs by sharing resources and achieving scalability. The words "single" and "instance" have been deliberately avoided, such that a qualifier can be used to determine whether we are speaking of single-instance or multiple-instance. The definition prescribes that when someone assigns the property multi-tenant, it is assigned to a system, service, database, or hardware resource, to clarify on what layer the multi-tenancy aspect applies. Although a small detail, it must be noted that multi-tenancy is written with a dash in 75% of the definitions.

There are several clarifications that can be made with the definition at hand. First, the word "transparently" refers to the fact that it is generally unknown to customers and end-users that another customer or end-user is using the same resources, otherwise the definition would be applicable to any web application that is open to multiple users (Google.com, Facebook, etc.).

A question that is frequently asked is what the differences are between multi-tenant, multi-user and multi-instance systems. The answer is that multi-instance systems do not necessarily need shared resources: a new system can be generated or deployed for each new user. Multi-tenant and multi-user systems, however, always share resources on one or more levels of the software stack. Multi-tenant systems share resources and allow only for mass-customization by using variability. Multi-user systems are only partly multi-tenant and offer the same invariable functionality to all customers. Please see Table 2.5 for an overview of these differences.

Multi	**Shared resources**	**Configurable at runtime**
-tenant	Yes	Fully
-user	Yes	Partly
-instance	Possibly	Possibly

TABLE 2.5: Difference overview for multi-tenant, multi-user, and multi-instance systems.

2.6 Research Agenda

In order to structure and guide future research in the area of multi-tenancy for both academics and practitioners, this section presents the major future research

topics identified in current research on multi-tenancy. The "future work" sections of all final papers identified in the systematic mapping studies were analyzed to extract potential future research topics. For this search all sections named "future work", "further work", "discussion" and "conclusion" were included. Also, all papers were searched entirely, using the keyword "future". First, all topics mentioned in the relevant sections were listed, after which synonyms and issues that were closely related were merged to overarching research themes. Classification and merging of the topics was performed by two researchers separately, after which the results were compared and discussed. This way, 23 issues were identified, which were categorized in four research themes. The analysis is based on the 48 papers that were collected in the structured mapping study. Every call for future work identified in the papers reflects a potentially strategic theme in the domain of multi-tenancy. Each of the themes below states the number of papers that address the theme and mention a specific call to action to researchers and practitioners.

Quality Assurance (6) — Compliance to Service Level Agreements (SLAs), performance, monitoring, all are mentioned in the current body of multi-tenancy literature as important issues to address in future research. Most issues within this topic are similar to important research challenges in the domain of SaaS (Zhang, Cheng, and Boutaba, 2010). This can be explained by the fact that multi-tenant software is always hosted in a SaaS environment, causing challenges in this domain to influence the multi-tenancy domain as well.
Call: An investigation into how customization of the multi-tenant application affects quality, e.g. in terms of performance. Can one general SLA be upheld, or should each tenant get a tenant-specific SLA?

Industry Validation (4) — Some papers reported on multi-tenant prototypes created, but all were missing a real validation. Because of this, a high number of papers call for industrial application of multi-tenant solutions. Applying prototypes in real industrial settings and performing more multi-tenancy related case studies can greatly enhance the validity of multi-tenancy research and is therefore considered to be a major topic in future research.
Call: With industrial multi-tenant solutions being developed right now, a next step for researchers is to work closely together with industry to validate research ideas on actual multi-tenant software systems.

Balancing & Placement (4) — Although all customers in a multi-tenant environment theoretically are served from one instance of a software product, in

practice, load balancing is needed between servers. This means identical servers are used to serve one software product in case this can no longer be done using one server. Specific tenants need to be placed on a specific server, but determining the best placement is a difficult task.
Call: There might be opportunities to develop better load balancing algorithms that take into account the historical usage of the application by the different tenants. Specifically, the load balancing can be targeted at looking at the different time zones in which the tenants are operating.

Database (4) — Four papers in the systematic mapping study explicitly mentioned database related issues as an important future research direction. Areas of interest include parallelism, locking, replication and partitioning.
Call: A major point of concern that we noted in the blog posts is data isolation, i.e., making sure that the data of individual tenants is shielded for other tenants. As such, an investigation into how to isolate and partition the data is a logical next step. Additionally, developing tests to make sure that data isolation is working correctly is also an interesting avenue for future work.

Three additional themes were identified, but were not sufficiently highlighted to count towards a valid collection of research themes. Although these themes were not emphasized by a sufficient number of authors, we mention them here briefly, to provide insight into other issues that are relevant. First, two papers mention the development of and research on multi-tenant platforms as an important next step in multi-tenancy research. The ***development of a multi-tenant platform (2)*** enables other researchers and developers to more easily deploy and test multi-tenant applications. Such a platform (ie. Salesforce (Fisher, 2007)) is likely to stimulate multi-tenancy research and development. The *call* in this context would be the need for an open platform available for multi-tenant applications. Researchers and industry should work together in designing, developing, and maintaining such a platform. Secondly, ***security (2)*** is a recurring theme in future work (Zhang, Cheng, and Boutaba, 2010), where papers specifically focus on the fact that different organizations, each having their own confidential data, are typically deployed on the same server and use the same instance of a software product. This increases the risk of data accidentally being queried by the wrong tenant. This leads to a *call* for increased attention to security in multi-tenant systems than it already does in multi-instance and multi-user systems. Finally, a theme that only occurs once in the literature that we surveyed, but poses a relevant challenge is ***variability (1).*** Since multi-tenant software is almost exclusively used

in a setting in which multiple different organizations use the same instance of a software product, variability is an important research topic. Variability is the ability of a software product to offer different configurations to organizations hosted on one instance of a software product. The definition of multi-tenancy presented in this paper also mentions 'varying customers', inducing the need for variability (Kabbedijk and Jansen, 2012). The corresponding *call* is that there should be more awareness on the importance of variability in multi-tenant software.

2.7 Threats to Validity

Since conducting a systematic mapping study is a largely manual task, most threats to validity relate to the possibility of researcher bias, and thus to the concern that other researchers might come to different results and conclusions. One remedy we adopted is to follow, where possible, guidelines on conducting systematic mapping studies as suggested by Budgen, Turner, Brereton, and Kitchenham (2008) and Petersen, Feldt, Mujtaba, and Mattsson (2008). The question of whether an article or blog post should be included in the mapping study is sometimes debatable. Following the advice of Kitchenham (2004), we enforced this criterion by utilizing predefined selection criteria that clearly define the scope (also see Section 5.2).

A potential threat to the validity of the interpretation of the results is researcher bias in the selection and filtering of the articles and blog posts. Our countermeasures were (1) the systematic logging of all data related to the screening and filtering steps in a database accessible by all authors of the paper and (2) randomly selecting 10% of all papers after each selection or filtering step to determine the inter-rater agreement for that subset of papers. If a paper is rated differently by another author, the discrepancy was discussed. Finally, this research assessed results published up to 2012, so the landscape of multi-tenancy could have evolved slightly in the mean time. This is identified as a threat to validity.

2.8 Conclusion

A total of 761 research papers and 371 industrial blogs on multi-tenancy have been analyzed in order to get a complete overview of the multi-tenancy domain. The results show that most papers propose a solution related to multi-tenancy,

but almost no papers report on industrial experiences while implementing multi-tenancy, providing some insight into the maturity of the domain. The blog analysis shows multi-tenancy is a popular topic and most blogs are written by individuals instead of corporations. Based on the research results a comprehensive definition for multi-tenancy is proposed **(SubRQ1)**, positioning multi-tenancy as an architectural principle of a system where multiple varying customers and their end-users transparently share the system's services, applications, databases, or hardware resources, with the aim of lowering costs. We call for this definition to be used in future research on multi-tenancy to further structure results and communication. No clear difference on the interpretation of multi-tenancy *between* academia and industry was observed, but we did see a significant difference *among* academia and industry **(SubRQ2)**. For future research we listed 4 themes **(SubRQ3)**, meant for the guidance of future research and providing a roadmap within the domain of multi-tenancy. The main research question **(RQ)** is answered by the complete drawing of the current multi-tenancy domain from both the academic and industrial perspective, together with the directions for steering the domain from this point on.

Chapter 3

The Role of Variability Patterns

Abstract

Within the business software domain, it is crucial for a software vendor to comply to different customer requirements. Traditionally this could be done by offering different products to different customers, but because multi-tenant business software deployments use one software product to serve all customers, this is no longer possible. Software vendors have to make sure that one instance of a software product is variable enough to support all different requirements from all different customers. This ability is defined as tenant-based variability.

Within this paper a conceptual model is presented, explaining the role software patterns play in solving variability implementation problems in multi-tenant business software. Different important aspects of patterns are explained, like forces and consequences and are linked to concepts in the problem domain. The paper suggests that variability patterns play a large role in addressing variability in multi-tenant business software and provide a valuable vocabulary for researching, reporting, thinking and communicating about variability solutions in online software products.

This work has been published as *The Role of Variability Patterns in Multi-tenant Business Software* in the proceedings of the WICSA/ECSA 2012 Companion Volume (Kabbedijk and Jansen, 2012). It is co-authored by Slinger Jansen.

3.1 Introduction

Within business software a frequently studied shift can be observed (D'souza, Kabbedijk, Seo, Jansen, and Brinkkemper, 2012) in which software products are no longer delivered to customers and deployed on-site *(on-premises)*, but are deployed at a central location and offered to customers online *(SaaS)*. Using the on-premises deployment method, one instance (i.e. one running copy of the software) of a software product is used by one customer. The product can be *tailored* or *customized* to comply to specific customer requirements in case the standard product functionality does not align with the requirements needed by the customer in order to support the business processes in place (Sun, Zhang, Guo, Sun, and Su, 2008). Another way to satisfy specific customer requirements is to create different products, based on a software product core containing all general requirements shared by all customers, which are generated in a software product line (Pohl, Böckle, and Linden, 2005). The ability to create different software products based on one software product core is referred to as variability and is defined as "Variability in software architectures describes how well an architecture supports flexibility in a certain aspect, with an exact specification of the differences." (Galster and Avgeriou, 2011, p. 63)

Software variability was first studied in software product lines. An attempt to give a complete taxonomy of variability in software products was done by Svahnberg, Gurp, and Bosch (2005), but this overview is focussed on software product lines and omits to take online software products into account. Traditionally, variability only takes on-premises software into account, which leads the fact software products containing specific features have to be created before shipping the software (i.e. early binding time). Online software, however, only profits from *run-time* binding times since it would be undesirable to restart or redeploy a software product whenever changes are made (as would be the case with design-time binding for example). Within online software products, variability is the result of the *configurability* of a product. The higher the configurability of a product is, the more variable a product is. A variable product aims to provide customers with "a multitude of options and variations using a single code base, such that it is possible for each tenant to have a unique software configuration" (Arya, Venkatesakumar, and Palaniswami, 2010).

Using the SaaS deployment model, it is still possible to have one instance of a software product per customer *(ASP)* (Tao, 2001), but because of the fact the

software instance is now deployed in a central location, multiple customers can potentially use the same instance. Sharing one instance of a software product with multiple customers, from a software vendor point of view, can be preferred above having separate instances because of (among others) economies of scale, easier data sharing, lower maintenance costs and improved scalability. Sharing a software instance with multiple customers, however, makes it impossible to have customized software products for a specific customer. In other words, MT software should be able to fulfil all different customer requirements while still profiting from shared resources. Whenever multiple customers share one instance of a software product and are able to configure the application to meet their requirements, we refer to the application as a *multi-tenant* SaaS application (Bezemer and Zaidman, 2010). Tenants are organizational entities (customers) using the application and usually consisting of multiple users. These tenants should be able to configure the product in the way they want because of this, the product should have the right level of variability at the right places.

Question remains however how to implement this variability in an efficient way. Within software architecture, specific problems are often solved in the same way. If the solution occurs a lot and is also a good solution to the problem, the solution can be documented in a structured way. These documented solutions to reoccurring problems are called software patterns. Within this research, software patterns are considered crucial in communicating variability implementation solutions in multi-tenant SaaS applications since they offer a structured and documented manner to do this. The goal of this paper is to explain the importance of variability in multi-tenant business software and explain the role software patterns play in achieving this variability. A model containing all different concepts and relationships between them will be presented in section 3.3.

3.2 Concepts

3.2.1 Tenant-based Variability

As indicated in the previous section, there is a need for variability in multi-tenant business software products. An important difference is made in variability management between *internal* and *external* variability (Pohl, Böckle, and Linden, 2005). The difference between these two variability types is crucial for defining tenant-based variability and relating it to the current variability concepts.

- **Internal variability -** Variability within the product that is only visible to the developers and architects. It is often an effect of different technical issues and standards. Having different methods for authenticating credit card transactions is an example of internal variability.
- **External variability -** This is the kind of variability that is related to different customers active within the product environment. If a customer, for example, wants to have a specific way of importing data, external variability needs to be in place to facilitate this.

The concept of external run-time variability is closely related to tenant-based variability, but since external runtime variability is related to customers in a *product line* context, it is not geared towards online (SaaS) software. The fact that in multi-tenant SaaS products, all customers share one software instance, causes changes in the standard way of thought related to external variability and adapt it to the online domain. An effort to use the concept of external runtime variability in a SaaS context is done by Mietzner, Unger, Titze, and Leymann (2009), who refer to this type of variability as *Customer-driven variability*. The concept of *tenant-based variability* used in this research is closely related to customer-driven variability, with an emphasis on multi-tenant software products. Different customers have different requirements to a software product. These differences require that the online software product is configurable to allow for the varying requirements.

3.2.2 Variability Patterns

Tenant-based variability in a software product can be implemented in several different ways. The appropriate solution depends heavily on the exact problem that needs to be solved. A good way of solving a specific problem is to apply a pattern related to this problem. Patterns are defined as "A particular recurring design problem that arises in specific design contexts, and presents a well-proven generic scheme for its solution. The solution scheme is specified by describing its constituent components, their responsibilities and relationships, and the ways in which they collaborate." (Buschmann, Meunier, Rohnert, Sommerlad, and Stal, 1996, p. 8)

A common mistake related to software patterns is the thought that they are a highly technical representation of a certain design choice, only readable and understandable by programmers. Patterns are far more general than that and can

be documented in a few different ways. Fowler listed a few common pattern description forms like for example *Alexandrian*, *GoF* or *POSA* style (Fowler, 1997). A combination of the different description forms can be used depending on the specific patterns. The forms are not prescriptive and should be used as guidelines in writing a pattern instead of set-in-stone rules.

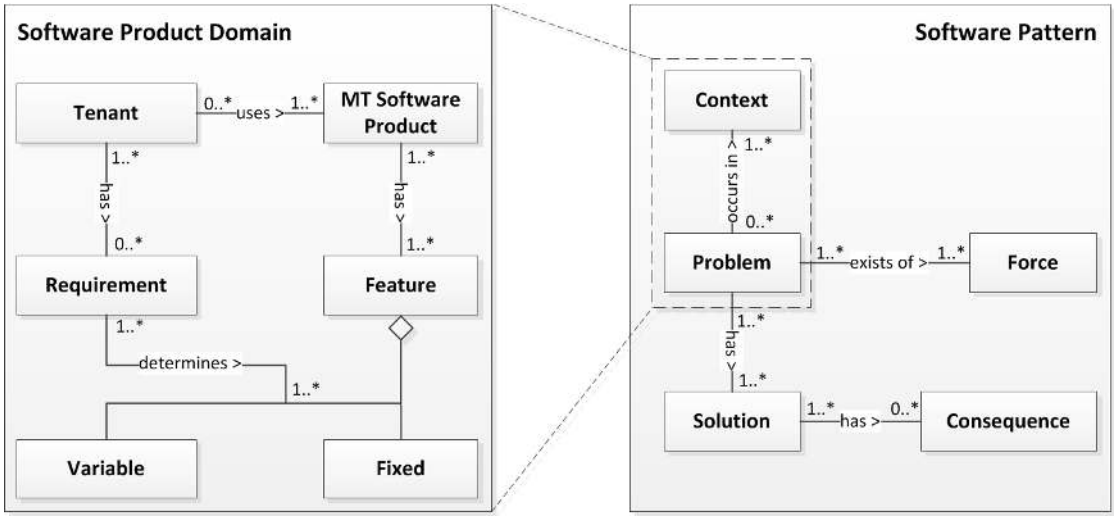


FIGURE 3.1: Conceptual Model: The role of variability patterns in multi-tenant business software

3.3 Conceptual Model

This section shows the relationships between the different concepts that play an important role within this research domain. Figure 3.1 shows all concepts as squares, connected by lines indicating the relationships. The left of the figure shows the problem domain, while the right of the figure shows the attributes of a software pattern that can be used to assess the problem. In the *Software Product Domain*, the *Multi-Tenant Software Product (MTSP)* plays a central role. The MTSP is used by different *tenants*, all having their own specific *requirements*. The MTSP contains certain functionality, indicated as *features* in figure 3.1. Features can be *fixed*, like, for example, the functionality of a software product to display financial data in a spreadsheet. Features can also be *variable* and influenced by the specific preferences of a tenant. Variable features could be the possibility to adapt the workflow within the MTSP or the ability to add specific data entities.

The *Software Pattern* contains a *problem* occurring in a specific *context*. The problem and context are related to the software product domain since this is the area of study. The problem and context in the domain of tenant-based variability

will always have to do with tenants having specific requirements and because of this the need for a variable multi-tenant online business software product. A specific problem always exists of different drivers that are the cause of the problem. These drivers are defined as *forces* within a pattern context (Buschmann, Henney, and Schmidt, 2007a). By making all the forces that are part of a problem explicit, and placing the problem in a properly defined context, a specific and useful *solution* can be documented. The implementation of a solution always has certain *consequences*, either advantages or liabilities (Schmidt, 1995). The pattern description used is based on Wellhausen and Fießer (2011).

The conceptual model presented in figure 3.1 identifies the most important concepts and related to multi-tenant business software and shows how software patterns can be used to answer and describe the problems related to the realization of variability in these systems. The conceptual model is a tool for answering problems related to tenant-based variability in multi-tenant software products that helps by creating a common lexicon for communicating solutions between and within industry and academia.

3.3.1 Application Example

As a real case example to illustrate the use of the conceptual model, an online supply chain management system is analyzed (*online software product*). The product is used by more than 120 customers of different sizes from all over the world (*tenants*), and more than 20.000 transactions are handled per customer per day. The MTSP supports the customers in tracking and planning different shipments and integrate different warehousing systems. In order to streamline the shipping process, some customers want to send a text message to a truck driver when and order is packed, but other customers first want a manager to check to order before it can be picked up. This means the two customers have different *requirements* regarding the workflow of the MTSP. The MTSP has a fixed way of handling workflows (*fixed feature)*, but want to be able to serve bot customers with their product (*problem*).

The identified *problem* in this *context* is caused by the fact customers have different business processes (*force*) and only one instance of the software is deployed on the server that is used by all tenants (*another force*). The *solution* that is applied by the MTSP vendor is the use of a component in the software product, capable of calling tenant-dependent components just before or after a certain action in the

workflow is performed. By doing this, the workflow within the MTSP becomes variable in such a way that different tenants can have different actions performed by the MTSP at certain time (*consequence*). The solution however also causes the MTSP to contain additional tables listing the tenant-dependent modules and checking for possible modules before and after each step in the workflow. This may cause performance issues if system load is high (*another consequence*). The make this pattern complete, a name (e.g. pre/post update hooks) and a clear diagram of the solution has to be added, but since this only a illustrative example this has been omitted for the sake of brevity.

3.4 Discussion

There is more to patterns than only the definition given by Gamma, Helm, Johnson, and Vlissides (1995) on patterns being "a solution". Besides being a solution, patterns also need to have a certain **"goodness"** and **recurrence** before they can be considered a pattern. The recurrence of a solution can be shown by performing a large number of case studies and reporting on the times the specific pattern is observed. A high recurrence, however, does not necessarily make a solution a pattern. Whenever a solution is frequently observed, but the solution has serious liabilities, the solution could even be considered an *anti pattern* (Brown, Malveau, and Mowbray, 1998).

Assessing the goodness of a pattern is difficult. Whether a solution is good or bad depends fully on the context of the problem. Scalability may, for example, be no issue at all for a software product aimed at five customers, while it is of the highest importance for a large ERP product aimed at thousands of SMEs. A way to still say something about the goodness of a pattern is to be thorough in identifying all the forces playing a role within the problem domain. If the forces are clearly listed, the consequences of applying the pattern can be related to the forces. The better all forces are handled, the higher the probability the pattern is a good solution. Possible liabilities should be analysed and, whenever possible, solutions for mitigating the liabilities should be given. Because of the generalized, implementation independent character of a software pattern, the goodness of a pattern can never be *validated*. However, by being complete in identifying forces and consequences, the goodness of a pattern can be evaluated.

Although the method used to describe a pattern can differ per pattern, depending on the context of the pattern (as discussed in section 3.2.2), patterns do need a

certain fixed form in order to compare different solutions to one another. For this research a form is chosen in which a pattern is always introduced by a **name**, a **context description** and a extensive description of the **problem**, including all **forces**. The solution is always introduced by a **diagram** (in no particular modelling language), an **explanation** and at least one real case **example**. After that all **consequences** (both advantages and liabilities) are assessed. Different patterns can be identified to solve a similar problem. In order to be able to compare the different patterns, the implementation consequences have to be documented in a structured way as well. Evaluating architecture quality attributes already documented in assessment standards (e.g. ISO/IEC 9126(ISO/IEC, 2011)) can be a solution to this, but only relevant attributes should be selected. Using this description form, together with a balanced set of quality attributes, gives a complete overview of a variability pattern and can be used in an efficient way for both research, educational and professional purposes.

3.5 Conclusion

Because different tenants in a multi-tenant environment should have the feeling they are the only one using a specific software product, meeting specific wishes of customers is crucial for a SaaS supplier. The software product needs to have a level of variability in order to achieve the configurability needed to meet those requirements. This paper proposes the use of software patterns to help implementing variability in an online software product. When reported on in a thorough way, software patterns are the ideal tool to report on variability solutions. The use of patterns compels to study forces and consequences of a solution in a structured way, enhancing the rigour of developing the solution. Since the credibility of a pattern description largely depends on the forces identified and the reported consequences, a fixed pattern form is recommended.

Overall, variability patterns play a large role in addressing variability in multi-tenant environments and are a valuable method for researching, reporting, thinking and communicating about variability solutions in online software products.

Chapter 4

Variability in Multi-tenant Systems

Abstract

In order to serve a lot of different customers in a SaaS environment, software vendors have to comply to a range of different varying requirements in their software product. Because of these varying requirements and the large number of customers, a variable multi-tenant solution is needed to achieve this goal. This paper gives a pragmatic approach to the concepts of multi-tenancy and variability in SaaS environments and proposes three architectural patterns that support variability in multi-tenant SaaS environments. The *Customizable Data Views* pattern, the *Module Dependent Menu* pattern and the *Pre/Post Update Hooks* pattern are explained and shown as good practices for applying variability in a multi-tenant SaaS environment. All patterns are based on case studies performed at two large software vendors in the Netherlands, who are offering an ERP software product as a service.

This work has been published as *Variability in Multi-tenant Environments: Architectural Design Patterns from Industry* in the Proceedings of the 30th International Conference on Advances in Conceptual Modeling (ER'11) (Kabbedijk and Jansen, 2011). It is co-authored by Slinger Jansen.

4.1 Introduction

Increasingly, product software vendors want to offer their product as a service to their customers (Ma, 2007). This principle is referred to in literature as SaaS (Gold, Mohan, Knight, and Munro, 2004). Turning software into a service from a vendor's point of view means separating the possession and ownership of software from its use. Software is still maintained and deployed by the vendor, but used by the customer. The problem of moving a software product from different on-premises locations to one central location, is the fact that it becomes really difficult to comply to specific customer wishes. In order to serve different customers' wishes, variability in a software product is needed to offer specific functionality. By making use of variability in a software product, it is possible to supply software functionality as optional modules that can be added to the product at runtime. Applying this principle can overcome many current limitations concerning software use, deployment, maintenance and evolution in a SaaS context (Turner, Budgen, and Brereton, 2003). It also reduces support costs, as only a single instance of the software has to be maintained (Dubey and Wagle, 2007).

Besides complying to specific customer requirements, a software vendor should be able to offer a service to a large number of customers, each with their own requirement wishes, without running into scalability and configuration problems (Bezemer and Zaidman, 2010). The solution to this problem is the use of multi-tenancy within a SaaS product. Multi-tenancy can be seen as an architectural design pattern in which a single instance of a software product is run on the software vendors infrastructure, and multiple tenants access the same instance (Bezemer, Zaidman, Platzbeecker, Hurkmans, and Hart, 2010). It is one of the key competencies to achieve higher profit margins by leveraging the economy of scale (Guo, Sun, Huang, Wang, and Gao, 2007). In contrast to a model incorporating multiple users, multi-tenancy requires customizing the single instance according to the varying requirements among many customers (Kwok, Nguyen, and Lam, 2008). Currently, no well-documented techniques are available on how to realize the variability needed in multi-tenant SaaS environments.

First the research method is discussed in section 4.2, after which the most important concepts in this paper will be explained and discussed in section 4.3. Then, the trade-off between the level of variability needed and the number of customers

is discussed in section 4.4, followed by three architectural design patterns for variability in SaaS product in section 4.5. The paper ends with a conclusion and future research in section 4.6.

4.2 Research Approach

In this chapter, variability patterns employed within the products of two large ERP SaaS providers are observed. In order to do this, a literature study has been performed, in which the combinations of *variability, saas* and *variability, multi-tenancy* were used as keywords in Google Scholar to get an overview of current variability patterns described in literature. Google Scholar is chosen as search engine since it indexes and searches almost all academic publishers and repositories world-wide. Papers resulting from the search are selected and put into a database if the keywords are mentioned in the title or the abstract. A total of *27* papers was collected during the search.

Besides the literature study, two independent case studies are performed at large ERP providers who recently launched their ERP software product as a service through the Internet (referred to as ErpCompA and ErpCompB from here on). ErpCompA has a turnover of around 250 million euros and around 20,000 users using their online product while ErpCompB has a turnover of around 50 million euros and around 10,000 users. The case studies were performed using the case study research approach by Yin (2009) and have a holistic multiple case design (cf. 1.4.2). The variability patterns are presented as architectural design patterns and created based on the Design Science principles of Hevner, March, Park, and Ram (2004), in which a constant design cycle consisting of the construction and evaluation of the variability patterns takes place. The initial model is constructed using an Exploratory Focus Group (EFG) (Hevner and Chatterjee, 2010), consisting of participants from academia and the case companies and a systematic literature review (Cooper, 1998). The participants in the EFG have been selected based on their experience in the area of variable multi-tenant SaaS-environments.

4.2.1 Validation

The validity of the patterns, identified in this chapter, is ensured by using multiple sources of evidence in a holistic multiple case design. The focus groups, used within the case companies, allow us to refine and validate our results constantly during

the research process. Additionally, draft versions of the variability patterns are discussed in interviews with key informants (i.e. software architects) within the two case companies (Runeson and Höst, 2009). Before patterns are constructed, the problem and related solution has to be observed in products from both case companies. The matching of the patterns enhances the internal validity (Yin, 2009). To ensure the external validity, all patterns are compared to the papers in the paper database. If similar patterns exist, the identified pattern is checked on completeness. The case study protocol was applied throughout the entire data gathering process.

4.3 Related Work and Definitions

To explain the multi-faceted concepts used in this paper, this section will discuss *multi-tenancy*, *design patterns* and *variability* in more depth. The definition proposed are meant to enable researchers to have one shared lexicon on the topic of multi-tenancy and variability.

4.3.1 Multi-tenancy

Multi-tenancy can be defined as "a property of a system where multiple customers, so-called tenants, have the possibility to configure the system; it allows them to transparently share the system's services, applications, databases, or hardware resources, with the aim of lowering costs" (cf. Chapter 2). A tenant refers to an organization or part of an organization with their own specific requirements, renting the software product. We define different levels of multi-tenancy:

- **Data Model Multi-tenancy**: All tenants share the same database. All data is typically provided with a tenant-specific Globally Unique Identifier (GUID) in order to keep all data separate. Even better is native support for multi-tenancy in the database management system (Schiller, Schiller, Brodt, and Mitschang, 2011).
- **Application Multi-tenancy**: Besides sharing the same database, all tenants also share the same instance of the software product. In practice, this could also mean a couple of duplications of the same instance, coupled together with a tenant load balancer (Kwok, Nguyen, and Lam, 2008).

- **Full Multi-tenancy**: All tenants share the same database and software instances. They can also have their own variant of the product, based on their tenant requirements. This level of multi-tenancy adds variability to the software product.

All items above are sorted on ascending implementation complexity.

4.3.2 Variability

The concept of variability comes from the car industry, in which different combinations of for example chassis, engine and color were defined as different *variants*. In software the concept is first introduced in the area of software product lines (Pohl, Böckle, and Linden, 2005), in which variability is defined as "the ability of a software system or artefact to be efficiently extended, changed, customized or configured for use in a particular context" (Svahnberg, Gurp, and Bosch, 2005). Within the area of software product lines, software is developed by the software vendor and then shipped to the customer to be run on-premises. This means variants have to be compiled before product shipping. Within the area of SaaS, software is still developed by the software vendor, but the product is served to all customers through the internet from one central place (Turner, Budgen, and Brereton, 2003; Kwok, Nguyen, and Lam, 2008). In principle, all variants can be composed the moment customers ask for some specific functionality, so at run-time. In this research, we define variability as follows: "Variability in software architectures describes how well an architecture supports flexibility in a certain aspect, with an exact specification of the differences." (Galster and Avgeriou, 2011, p. 63)

We identify two different types of variability within multi-tenant SaaS deployments:

- **Segment Variability**: Product variability based on the segment a tenant is part of. Examples of such variability issues are different standard currencies or tax rules per country or a different layout for Small and Medium Enterprises (SMEs) and sole proprietorships.
- **Tenant-oriented Variability**: Product variability based on the specific requirements of a tenant. Examples of such variability issues are different background colors or specific functionality.

We also identify different levels of variability in tenant oriented variability:

- **Low: Look and Feel**: Changes only influencing the visual representation of the product. These changes only occur in the presentation tier (tier-based architecture (Eckerson, 1995)) or view element (MVC-based architecture (Krasner and Pope, 1988)). Examples include different background colors or different element sorting in lists.
- **Medium: Feature**: Changes influencing the logic tier in tier-based architecture or the model or controller element in an MVC-based architecture. Examples include the changes in workflow or the addition of specific functionality.
- **High: Full**: Variability of this level can influence multiple tiers at the same time and can be specific. Examples of this level of variability includes the ability for tenant to run their own program code.

The scope of this research is focused on *runtime tenant-oriented low and medium variability in multi-tenant enterprise software.*

4.3.3 Software Patterns

The concept of patterns was first introduced by Christopher Alexander in his book about the architecture of towns (Alexander, Ishikawa, Silverstein, Jacobson, Fiksdahl-King, and Angel, 1977). This concept was quickly picked up in the software engineering world and led to the famous 'Gang of Four' pattern book by Gamma, Helm, Johnson, and Vlissides (1995). This book describes several patterns that are still used today and does this in a way that inspired a lot of subsequent pattern authors. The definition of a pattern used in this paper originates from the Pattern Oriented Software Architecture series (Buschmann, Meunier, Rohnert, Sommerlad, and Stal, 1996; Schmidt, Stal, Rohnert, and Buschmann, 2000; Kircher and Jain, 2004; Buschmann, Henney, and Schmidt, 2007a; Buschmann, Henney, and Schmidt, 2007b) and reads: "A particular recurring design problem that arises in specific design contexts, and presents a well-proven generic scheme for its solution. The solution scheme is specified by describing its constituent components, their responsibilities and relationships, and the ways in which they collaborate." (Buschmann, Meunier, Rohnert, Sommerlad, and Stal, 1996, p. 8)

Patterns are not artificially created artifacts but evolve from best practices and experiences. The patterns described in this paper result from several case studies and discussions with experienced software architects. All patterns have proven

to be a suitable solution for the problems described in section 4.5, since they are applied in successful SaaS products at the case companies. Also, all patterns are described language or platform independent, so the solution can be applied in various situations in the SaaS domain. More information on future research concerning the patterns proposed can be found in section 4.6.

4.4 User-Variability Trade-off

The best solution for deploying a software product from a software vendor's perspective depends on the level of resources shared and the level of variability needed to keep all users satisfied. In figure 4.1 four deployment solutions are introduced, that are considered best practices in the specific situations shown. In this section, the need for multi-tenant deployment models is explained.

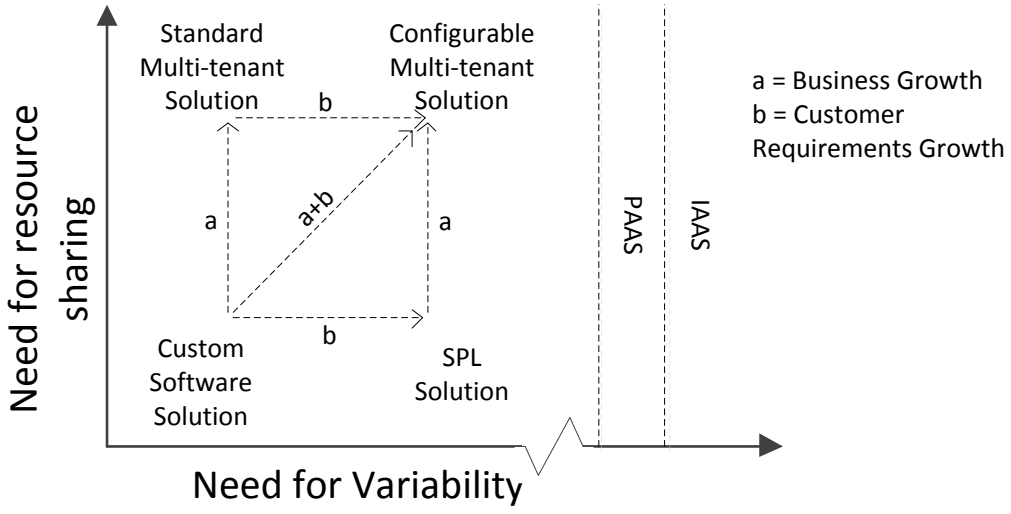


FIGURE 4.1: Level of variability versus Number of users

By using the model shown in figure 4.1, software vendors can determine the best-suited software deployment option. On the horizontal axis, the need for variability in a software product is depicted, and the number of customers is shown on the vertical axis. For a small software vendor who does not have a lot of customers with specific wishes, a standard *custom software solution* is sufficient. The more customers software vendors get (business growth), the higher the need for a *standard multi-tenant solution* because of the advantages in maintenance. When the amount of specific customer wishes grows, software vendors can choose the SPL

approach to create variants for all customers having specific requirements. This solution can lead to a lot of extra maintenance issues as the number of customers grows. In case of a large number of customers having specific requirements, a *configurable multi-tenant solution* is the best solution for software vendors, keeping an eye on performance and maintenance.

4.5 Variability Patterns

In this section, three patterns are described that were observed in the case studies that were conducted. The patterns are based on solutions observed within the case companies' software product, refined by patterns already documented in literature (Gamma, Helm, Johnson, and Vlissides, 1995; Svahnberg, Gurp, and Bosch, 2005). All patterns will be explained by a UML diagram, together with a pattern description based on the model described in Chapter 3. The model has been extended by an explanation and an example, in order to add more detail to the pattern description.

4.5.1 Customizable Data Views

In this section, a variability pattern is discussed, enabling developers to give tenants a way to indicate their preferences on the representation of data within the software product.

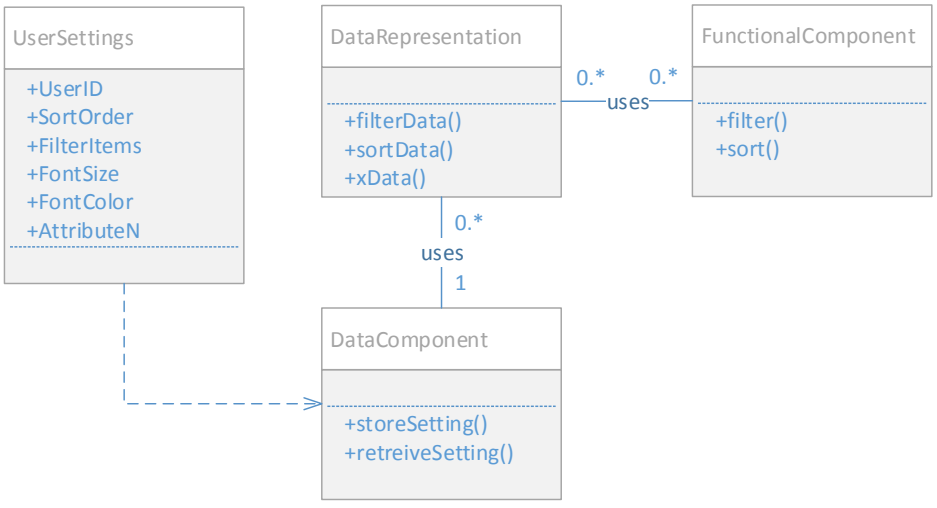


FIGURE 4.2: Customizable Data Views Pattern

Context — Design of a multi-tenant enterprise application.
Problem — It is important to give the tenant the ability to indicate and save his preferences on the representation of data shown. How can developers be enabled to give tenants a way to indicate their preferences on the representation of data within the software product?
Solution — In this variability pattern (cf. figure 4.2), the representation of data is performed at client side. Tenants can, for example, choose how they want to sort or filter their data, while the data-queries do not have to be adapted. The only change needed to a software product is the introduction of tenant-specific representation settings. In this table, all preferred font colors, sizes and sort option can be stored in order to retrieve this information on other occasions to display the data again, according to the tenant's wishes.
Explanation — As can be seen in the UML representation of the pattern in figure 4.2, the *DataRepresentation* class can manipulate the appearance of all data by making use of a *FunctionalComponent* able of sorting, filtering, etcetera. All settings are later stored by a *DataComponent* in a specific *UserSettings* table. Settings can later be retrieved by the same DataComponent, to be used again by the DataRepresentation class and FunctionalModule.
Consequences — By implementing this pattern, one extra table has to be implemented. Nothing changes in the way data selection queries have to be formatted. Representation of all data has to be formatted in a default way, except if a tenant changes this default way and stores his own preferences.
Example - In a bookkeeping program, a tenant, for example, can decide what columns he wants to display and how he wants to order them. By clicking the columns he wants to display, his preferences are saved in the database. When the tenant uses the product again later, his preferences are fetched from the database and applied to his data.

4.5.2 Module Dependent Menu

This section describes a pattern to create dynamic menus, based on the modules associated to a tenant.

Context — Design of a multi-tenant enterprise application.
Problem — All tenants have specific requirements to a software product, they can all use different sets of functionality. Displaying all possible functionality in the menu would decrease the user experience of tenants, so menus have to display only the functionality that is relevant to the tenant. How can a custom menu to

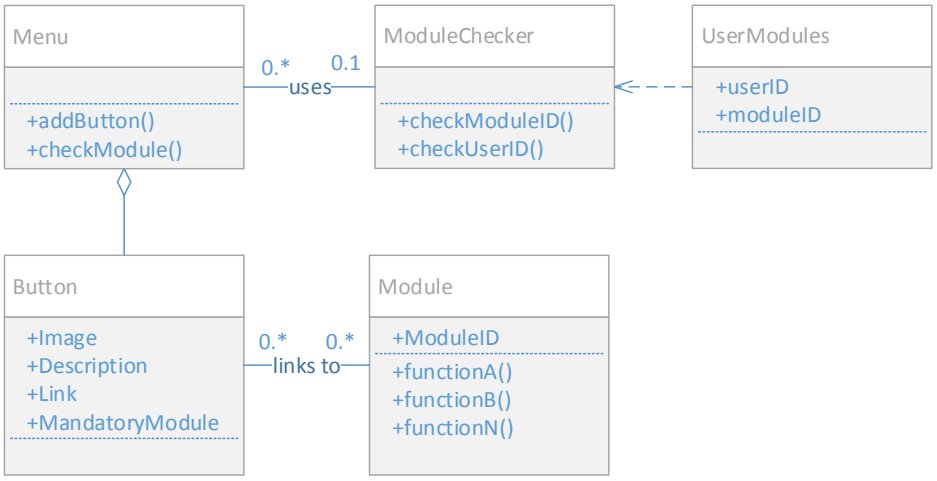


FIGURE 4.3: Module Dependent Menu Pattern

all tenants, only containing links to the functionality relevant to the tenant be provided?

Solution — The pattern proposed (cf. figure 4.3) creates a menu out of different buttons based on the modules associated to the tenant. Every time a tenant displays the menu, the menu is built dynamically based on the modules he has selected or bought.

Explanation — The *Menu* class aggregates and displays different *buttons*, containing a link a specific module and the prerequisite for displaying this link (*mandatoryModule*). The selection of buttons is done, based on the results of the *ModuleChecker*. This class checks whether an entry is available in the *UserModules* table, containing both the ID of the tenant (user) and the mandatory module. If an entry is present, the *Menu* aggregates and displays the button corresponding to this module.

Consequences — To be able to use this pattern, an extra table containing user IDs and the modules available to this user has to be implemented. Also, the extra class *ModuleChecker* has to be implemented. All buttons do need a notion of a mandatory module that can be checked by the ModuleChecker to verify if a tenant wants or can have a link to the specific functionality.

Example — In a large bookkeeping product, containing several modules that can be bought by a tenant, the menus presented to the tenant can be dynamically composed based on the tenant's license.‘

4.5.3 Pre/Post Update Hooks

In this section a pattern is described, capable of implementing modules just before or after a data update.

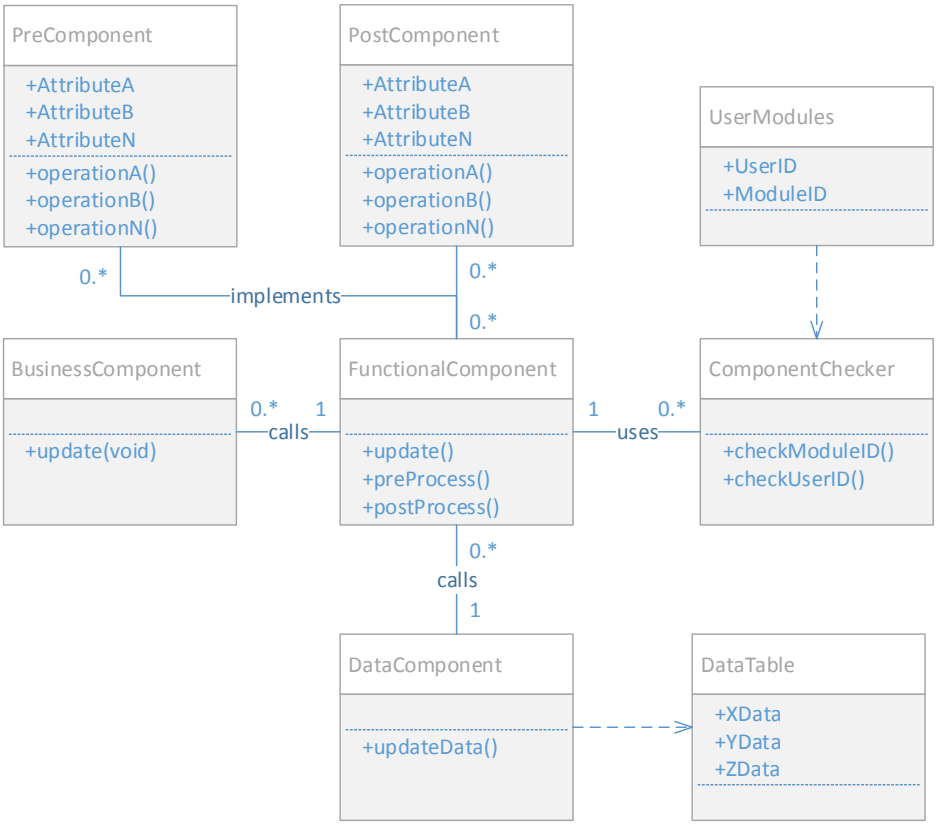


FIGURE 4.4: Pre/Post Update Hooks Pattern

Context — Design of a multi-tenant enterprise application.

Problem — In business oriented software, workflows often differ per tenant. To let the software product fit the tenant's business processes best, extra actions could be made available to tenants before or after an event is called. How can the possibility for tenants to have custom functionality just before or after an event be provided?

Solution — The pattern introduced here (cf. figure 4.4) makes use of a component able of calling other components before and after the update of data. The tenant-specific modules are listed in a separate table, similar to the pattern described in section 4.5.2.

Explanation — Before the *FunctionalComponent* calls the *BusinessComponent*

in order to perform an update, the *ComponentChecker* is used to check the *UserModules* table if a tenant wants and may implements an extra component before the update is performed. After this, the *BusinessComponent* is called and the update is performed. The *DataComponent* takes care of the writing of data to a specific data table. After this, the *ComponentChecker* again checks the *UserModules* table and a possible *PostComponent* is called.
Consequences — Extra optional components have to be available in the software system in order to be able to implement this pattern. The number of components available depends on the tenants' requirements.
Example — In a bookkeeping program, tenants can choose, whether they want to update a third party service as well by using a component that uses the API of a third party service to make changes there. If so, the FunctionalComponent can call a third party communicator after an internal update is requested.

4.6 Conclusion and Future Research

This paper gives a classification of different types of multi-tenancy and variability, enabling researchers to have one shared lexicon. Satisfying the need for a pragmatic overview on how to comply to the specific requirements of large numbers of customers, while keeping a product scalable and maintainable, this paper showed an introduction to the concept of variability in multi-tenant SaaS solutions and presented three patterns gathered from industry case studies. By applying these patterns, companies can better serve customers and keep their software product maintainable and scalable. All three patterns are proven to be effective within the case companies and are reviewed by experts from the case companies, but still need to be analyzed more in terms of performance and effectiveness.

More variability patterns still have to be identified, and the effectiveness, maintainability, scalability and performance of all variability patterns still has to be tested in future research. Currently, a preliminary variability pattern evaluation model is being developed enabling researchers to test all identified variability patterns and draw conclusions on their effectiveness. Also more case companies from other domains will be examined, enabling us to identify and test more variability patterns.

Chapter 5

Variability Consequences of the CQRS Pattern

Abstract

In order to maximize their customer base, business software vendors are trying to offer software products that support the business needs of as many customers as possible. The more standardized a software product is, the easier it will be to serve large numbers of uniform customers. However, if customers are not homogeneous, a trade-off must be made between flexibility and complexity. A case study is presented showing the implementation of the CQRS pattern, a pattern dictating the strict separation between commands and queries. The study was performed at a large software product vendor currently designing a software product based on CQRS. Seven sub patterns related to CQRS are identified and discussed. The research results show the CQRS pattern is implemented and how its different sub patterns can result in a high level of variability within a software product and how the different sub patterns can interact to achieve this.

This work has been published as *A Case Study of the Variability Consequences of the CQRS Pattern in Online Business Software* in the Proceedings of the 17th European conference on Pattern Languages of Programs (EuroPLoP 2012) (Kabbedijk, Jansen, and Brinkkemper, 2012). It is co-authored by Slinger Jansen and Sjaak Brinkkemper.

5.1 Introduction

It is highly relevant in business software to offer a product to customers that fits their business processes, especially in ERP and related bookkeeping software. This can be problematic, since different customers have different business processes and because of these different, or even contradictory, requirements to a software product. The architecture of a software product has to support the variability needed to offer a software product flexible enough to match all different customer requirements, while not introducing unwanted side effects, such as complexity, scalability or security challenges. In Software Product Lines (SPL), variability is known as the ability to change or customize a software product (Jaring and Bosch, 2002). This definition is sufficient for software products that are manufactured in a software product line style and deployed on premises at customers, but does not hold true anymore when it comes to online software products. Online software products have to be able to offer variable solutions *at the same time* from a single customizable instance, a concept known as runtime variability (Mietzner, Metzger, Leymann, and Pohl, 2009). The principle of serving multiple customers from one online software product, giving each the idea they are the only customer using the product in terms of flexibility is known as multi-tenancy (Kabbedijk and Jansen, 2011).

In order to create a software product, capable of offering a certain level of variability, most current software products separate logic into different layers. Each tier within this architectural principle is responsible for a different part of the architecture (Manuel and AlGhamdi, 2003). An often implemented solution to this multi-tier architecture is the three-tiered application in which there is a separate data, logic and presentation tier. Within this solution, the database in the data tier is often seen as one CRUD (Create, Read, Update and Delete data) data store in which all commands and queries are performed on the same database. This can lead to locking, performance and scalability problems, especially with larger commands or queries since all things have to be taken care of sequentially. Distributing parts of the system in combination with selective locking of data provides a partial solution but leads to a high probability of data inconsistency.

Since the CAP theorem (Gilbert and Lynch, 2002) states that it is impossible for a distributed system to have Consistency, Availability and Partition Tolerance at the same time, it is an option to split parts of the system that have an emphasis on consistency from parts that should have an emphasis on availability or partition

tolerance. Following this line of thought, Greg Young and Udi Dahan came up with the CQRS (Command Query Responsibility Separation) pattern (Young, 2010; Dahan, 2010) in which all logic of a software product is separated based on whether it changes the application state (commands) or only queries it. This means executing commands is done by different components than the one responsible for executing the querying tasks, all of which can be done distributed and in parallel.

Besides helping to solve the scalability problem of multi-tiered software products by enabling architects to distribute tasks of the system among an unlimited amount number of systems, CQRS also helps to implement a higher level of variability in online software products. The high level of variability is caused by the fact the main pattern keeps commands strictly separated from queries and has a large collection of sub patterns using the distributed nature of the pattern to enable, among others, all sort of different tenant dependent configurations, workflows, and business rules. This concept will be further explained in section 5.6.

This paper will first report on related research in section 5.2, after which the research approach will be discussed in section 5.3. After this, an example of the CQRS pattern will be shown we observed in the case company in section 5.4. Different sub patterns playing a role in CQRS will be discussed in section 5.5, followed by the consequences CQRS has in section 5.6. The paper ends with a discussion and some future research in section 5.7, followed by a conclusion in section 5.8.

5.2 Related Work

The ground principle of CQRS, stating the strict separation of command and queries, is introduced by Bertrand Meyer in his book Object-Oriented Software Construction (Meyer, 1988). He called it Command Query Separation (CQS), a pattern in which each method is either a command performing a certain action or a query returning data to the caller. Both commands and queries are performed independently from each other. In his own words, “asking a question should not change the answer”. This concept was picked up later on by Greg Young and Udi Dahan, who merged it with ideas out of Domain Driven Design (DDD) by Eric Evans (Evans, 2004) and combined this to create the CQRS pattern (Young, 2010).

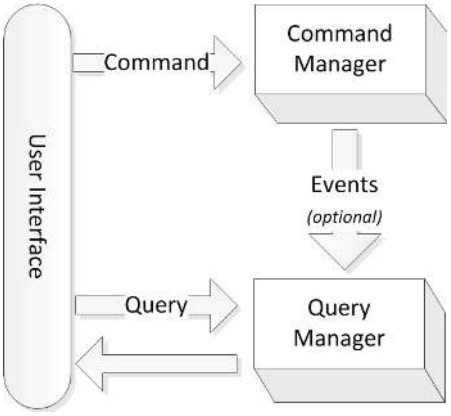


FIGURE 5.1: CQRS core principle

In a nutshell, the CQRS pattern is only about creating two subsystems, as can be seen in figure 5.1. From the user interfaces, commands can be sent to the command manager or queries can be sent to or received from the query manager. Commands are actions that will be performed on the data, while queries are requests for data to be shown. The CQRS pattern itself does not prescribe anything about communication between the command manager and the query manager, but there is a collection of patterns often used in combination with CQRS that take care of communication. An often applied pattern within CQRS for communication is communication through events, which will be elaborated on in section 5.5.1.

Currently there is an active community of developers, architects and enthusiasts working with the CQRS pattern, but it has not yet penetrated the broadly applied and widely known collection of software patterns described in the work of Gamma et al. (Gamma, Helm, Johnson, and Vlissides, 1995) and the Pattern-Oriented Software Architecture books (Buschmann, Meunier, Rohnert, Sommerlad, and Stal, 1996; Schmidt, Stal, Rohnert, and Buschmann, 2000; Kircher and Jain, 2004; Buschmann, Henney, and Schmidt, 2007a; Buschmann, Henney, and Schmidt, 2007b). Several frameworks like NCQRS (`github.com/ncqrs`), Axon (`github.com/axonframework`) and Lokad (`github.com/Lokad/lokad-cqrs`) however, helping developers to implement the CQRS pattern in several languages are released in the last two years, making the CQRS pattern increasingly popular. The documentation and community around these frameworks are also an important source of knowledge related to the CQRS pattern.

5.3 Research Approach

In order to gather the data relevant for this research, a case study was performed at large ERP software vendor from the Netherlands having approximately 10,000 small and medium enterprises using their current online bookkeeping software product on a day to day basis (i.e. ERPCompB, as discussed in Chapter 4). In this chapter, we will refer to the case company as ERPComp. During this case study, **(1)** several CQRS information sessions for employees at ERPComp in which the principles of CQRS and are explained, emphasizing on the particular implementation within ERPComp were attended. We **(2)** actively participated in the architecture team for a total of 40 hours in which we also **(2a)** conducted five interviews with architects working at ERPComp on the implementation of CQRS within their software product and its consequences on scalability and variability. Within these interviews, architecture design artifacts were discussed and shared with the author. Finally, **(2b)** all results of the interviews were analyzed within the architecture team to have a constant feedback loop for interpreting the architecture and consequences of implementing the CQRS pattern. This constant feedback of key figures within the research area is common practice within cooperative inquiry (Reason, 1994) and the design science cycle of Hevner, March, Park, and Ram (2004). Besides gathering data within ERPComp, **(3)** all research findings are also discussed with an external expert panel consisting out of three leading CQRS experts from outside the case company. All experts are either author of a CQRS framework or provide courses in applying the CQRS framework. The panel also actively participated in reviewing the entire chapter. The overview of the research approach used can be found in figure 5.2.

The study performed is a 'single case' case study according to the classification of Yin (Yin, 2009). A case study database containing recordings of the interviews and all notes taken during the interviews was kept in order to improve the traceability and rigour of the case study research. The internal and construct validity of the research is ensured by using experts within the case company to check all artifacts and conclusions and by matching the results of the case study with expected results. A clear case study protocol was used as advised by Runesson and Höst (Runeson and Höst, 2009) in which the planning and structure of the interviews was described.

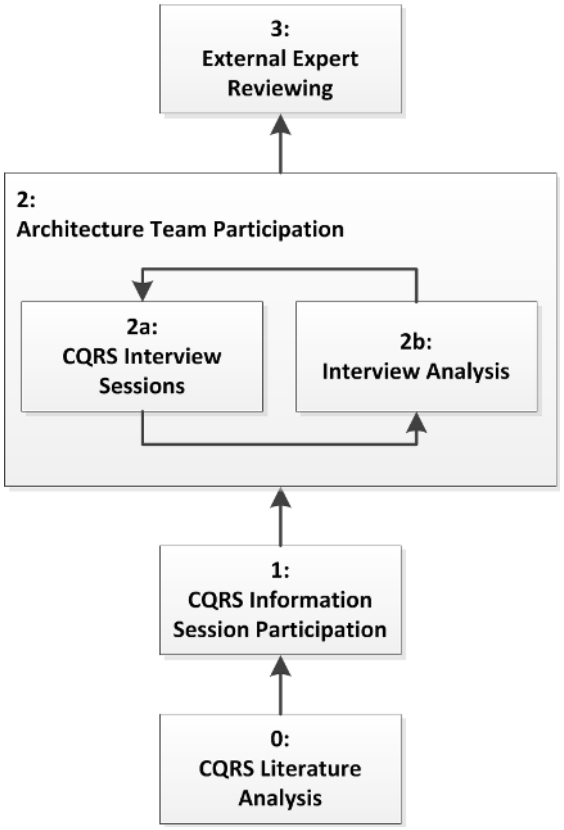


FIGURE 5.2: Applied Research Method

5.3.1 Research Questions

The main research goal **"How can the CQRS pattern influence the variability of a software product?"** will be answered by answering three related research questions (RQs). These questions are:

1. How is the CQRS pattern designed within the case company?
2. What sub patterns can be identified within the CQRS pattern?
3. How do the different sub patterns influence the variability of a software product?

RQ1 is answered by a using interview data and architectural design artifacts gathered at the case company during CQRS information sessions, interviews and constant feedback during architecture team participation. The answers to RQ2 are primarily answered by using design artifacts from the case company, combined with literature and expert reviews from an external expert panel consisting of three CQRS experts. RQ3 again is answered using the interview data resulting

from interviews held with architects at the case company and a review on our conclusions by an external expert panel.

5.3.2 Validation

Since we took part in the architecture team at ERPComp, the data resulting from the case study, are constantly validated by discussion within the team. The constant short feedback loop enhances the construct validity of the patterns identified (Reason, 1994). Additionally, all patterns identified within the case study, are explicitly discussed during five interviews with experts from ERPComp (enhancing internal validity) and compared to current documentation available online (enhancing external validity). The patterns are not only compared to academic literature, but also to industrial blogs and documentation, because the concept of CQRS is still young, and not many academic papers have been published yet on the topic. Also, three external CQRS consultants are used as experts to validate the CQRS patterns. The experts all, individually, reviewed the patterns, and feedback on the pattern content and presentation was processed accordingly. After the initial feedback, the improved patterns are validated once more to ensure the changes are performed correctly.

5.4 CQRS Implementation

This section will report on the design of the software architecture of the main product created by ERPComp. This implementation report is aimed at giving an impression of the possibilities of the CQRS pattern and related sub patterns. Currently, ERPComp is redesigning their software product from scratch, keeping a strict separation between all queries and commands as indicated by the CQRS pattern. This section reports on the new software architecture designed at ERPComp, so no legacy code or systems are in place.

Figure 5.3 shows the CQRS-based design of the software architecture at ERPComp. On the left side of the figure the *User Interface* is modelled from which *Commands* can be sent to the *Command Bus* (top of the figure) and *Queries* can be sent and received to and from the *Query Bus* (bottom of the figure). All arrows in the figure represent communication within the system. Whenever the communication is explicitly implemented as a command, query or event, this is indicated in the figure. All other arrows, including the double headed arrows only represent

a certain form of communication, but nothing specific is specified. From the *Command Bus*, commands are sent to *Command Handlers*. The handlers perform the action indicated by the command, after which the action is stored in the /textit-Stream Store. From here all events are sent to the *Event Bus*, who can distribute the event among different *Query Model Builders (QMBs)* or route events back to the *Command Bus*, through *Event Routers*. Add the query side, different *Query Model Builders* listen to the events broadcasted through the *Event Bus* and act on certain events. The events they response to depend on the goal of the specific builders. All built queries are stored in a *Query Store*, for easy access by both the *Query Model Builders* and *Query Handlers*. The *Query Handlers* can publish query results to the *Query Bus*, which can be used by he *User Interface* to display certain information.

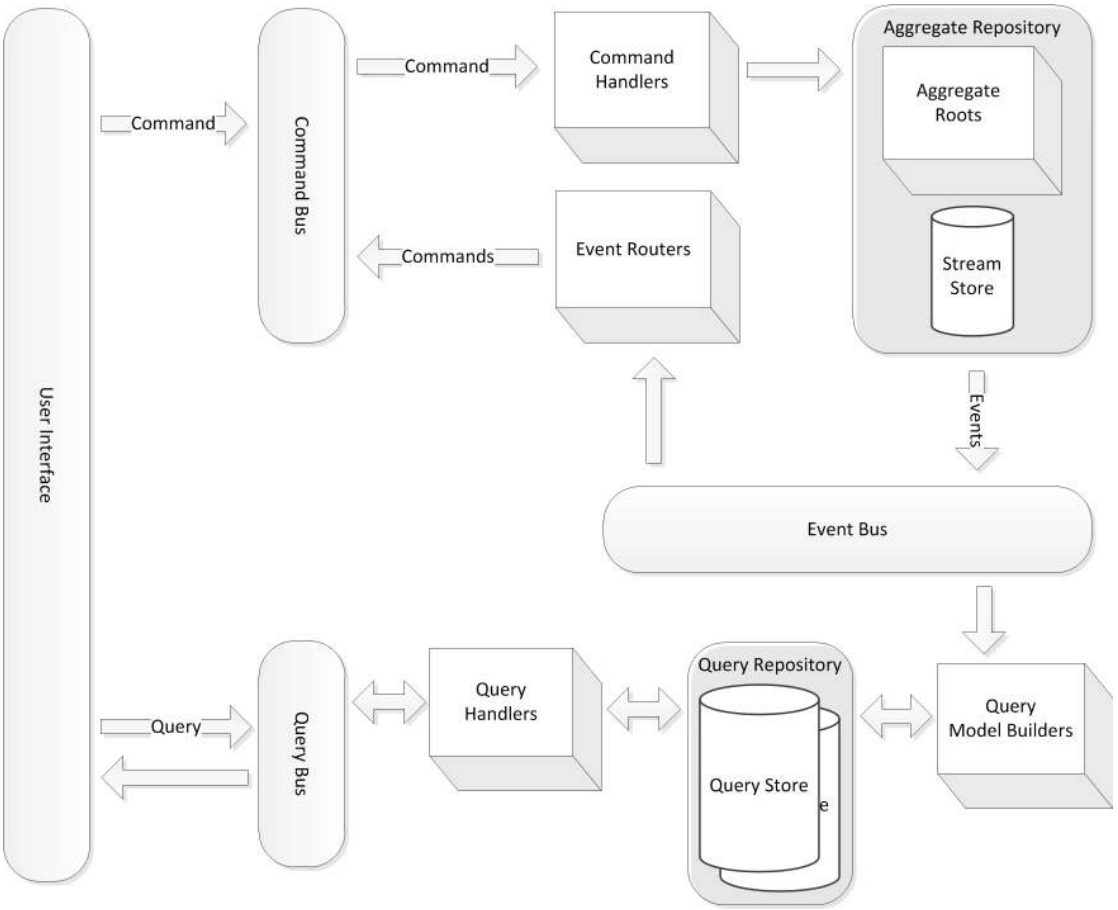


FIGURE 5.3: CQRS implementation at ERPComp

A further analysis of sub patterns that can be observed within the CQRS-based software architecture design can be found in section 5.5.

5.5 CQRS Sub Patterns

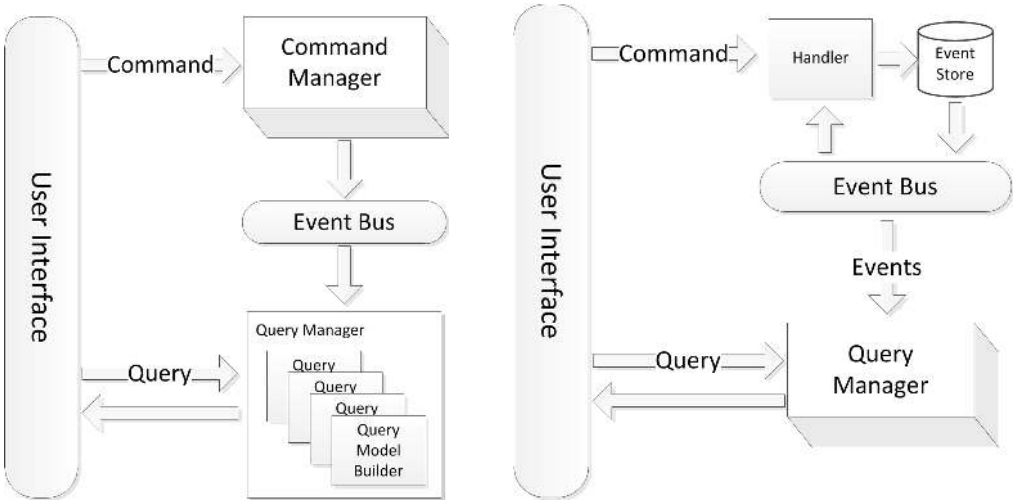
The CQRS pattern can be extended and complemented by applying several additional patterns. The number of sub patterns that can be applied within the CQRS pattern are numerous, but this research focusses on the selection of patterns we observed within the case company. All sub patterns are described using the pattern description model from Chapter 3. The *context* for all sub patterns is the design of a multi-tenant enterprise application, applying the CQRS Pattern.

5.5.1 Event Sourcing

Problem — There needs to be a way to communicate between the command manager and the query manager.
Solution — One of the possibilities within the CQRS pattern that can play a big role in terms of scalability is the sourcing of the events created by the command manager. These events can be sent to an event bus to which the query model builders in the query manager listen. A query model builder is a different sub pattern that is able of translating events to appropriate data views, which is discussed in more depth in section 5.5.5. The different query builders can be on the same system, but also on different physical or virtual machines. Query model builders can be on different geographical locations or even at clients. Because of this, the system becomes scalable, and all sub parts are especially geared towards the task they have to do (i.e. read or write) (Young, 2010). Please see figure 5.4a for a representation of the Event Sourcing pattern. The most important aspect of the event sourcing pattern is the fact different events are broadcasted by the command manager to be processed by different components.
Consequences — The system becomes scalable and all sub parts are especially geared towards the task they have to do (i.e. read or write).

5.5.2 Event Store

Problem — Storage by the query manager could be anything, from stored in cache to stored at the client, or in some database. Because of the uncertainty in storing method, you can not rely on the availability and recovery of data if the system crashes.
Solution — Events in the CQRS pattern do not necessarily need to be stored in any way. They could be sent to the query manager immediately after being



(A) Event Sourcing Sub Pattern (B) Event Store Sub Pattern

FIGURE 5.4: The Event Sourcing and Event Store CQRS sub patterns

processed and never be stored (as can be seen in figure 5.4a). The query manager then can do with the event whatever is necessary to get the data in an appropriate form. Because the storage by the query manager could be anything, from stored in cache, to stored at the client, or in some database, you can not rely on the availability and recovery of data if the system crashes. Because of this, an often implemented pattern within CQRS is the use of an event store. In this store, all events can be stored sequentially, so all data can be reconstructed based on the events in case of a system crash (Nijhof, 2010). Figure 5.4b shows the representation of the event store pattern. From the *User Interface* commands are sent to a *handler* (see section 5.5.4 for more information on command or query handlers) who sends it to the event store as an event.

Consequences — Events are now stored in a central location and can be accessed in a reliable way, for the sake of data recovery.

5.5.3 Aggregate Root

Problem — Data in the CQRS pattern is created by different query model builders and because of the asynchronous way the listeners work nothing is known about the correctness of data at the time of querying.

Solution — Because data in the CQRS pattern is created by different query model builders of which you do not necessarily know what or where they are, and because of the asynchronous way these listeners work you can not say anything about the correctness of data at the time of querying. As an example, think about a large web shop selling laptops. Whenever someone wants to order a laptop, the system

needs to know whether the inventory is sufficient to approve the order. In other words, the system needs to be sure there is at least one laptop available before the order can be processed. In the core CQRS pattern, there is no way to know for sure the laptop is in stock because all events are processed asynchronously. The only way to know for sure the laptop is in stock is to store the number of laptops available together with the laptop itself and also process this as one. If not, it is possible that the system checks whether a laptop is in stock, sees one laptop in stock, starts processing the order and ends up with an erroneous order since the laptop is sold just before through another process.

The concept of storing and processing all properties and entities that are dependent on each other together is know as aggregation. The main entity is called the entity root. An order, for example, should always be processed together with its order lines, since the lines make no sense without the order. In the previously mentioned example, the order and order lines are an *aggregate* and the order is the *aggregate root*, since deleting the root would indicate deleting the other entities as well.

Consequences — Related properties and entities are processed and stored together.

5.5.4 Command Handler

Problem — Commands coming in from the user interface have to be passed through to something that will perform the action dictated by the command.

Solution — Commands coming in from the user interface have to be passed through to something that will perform the action dictated by the command. As discussed in section 5.5.3, these actions can be adequately performed by aggregate roots as observed within the design of ERPComp. The command coming from the command bus has to be interpreted and translated somehow before it can be performed. A command handler is capable of catching one or more commands and passing it through to an object capable of performing the command (Abdullin, 2010). Figure 5.5 shows an overview of the command handler pattern. The action performer in the figure should somehow make sure an action is performed. One way of doing this is using a two phase commit (Gray and Lamport, 2006) in which a request is sent in two phases, but since this adds significant load to the system, other methods like delaying the sourcing of events until an aggregate root is totally finished are recommendable.

Consequences — Commands are correctly and timely processed.

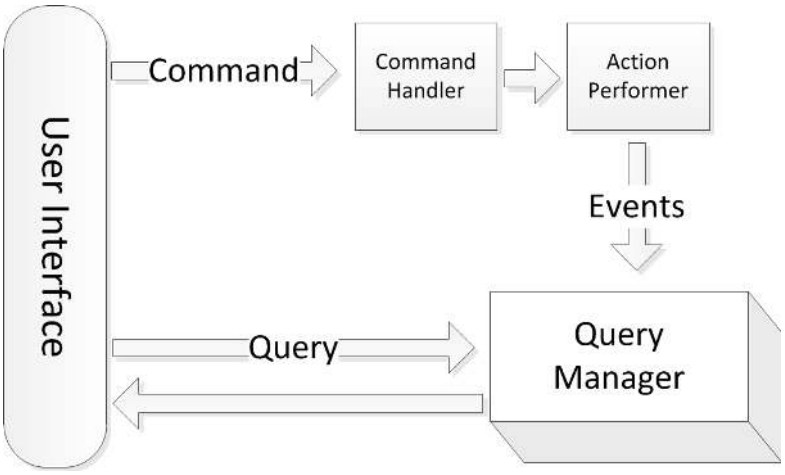


FIGURE 5.5: Command Handler Sub Pattern

5.5.5 Query Model Builder

Problem — Data queries by tenants are diverse and need to be translated to an appropriate view. In order to represent the right data in the appropriate form, the needed view is dependent on the domain of the query. The domain knowledge needs to be translated to an automatically usable model.
Solution — All events that are sent to the query manager can be caught by a query model builder, as discussed in section 5.4a. These query model builders can be everywhere from the client's cache to all kind of different physical servers. The QMBs listen to events coming in through the event bus and create a view of the data needed by the query manager. This view totally depends on the domain the QMB is in and the goal the data has. A QMB in a system responsible for generating inventories, for example, will build entirely different query models than a QMB in a system responsible a displaying the contact details of one person. Figure 5.6 shows a representation in combination with the query handler pattern discussed in section 5.5.6.
Consequences — Queries are translated to an automatically usable model.

5.5.6 Query Handler

Problem — Data queries by tenants are diverse and need to be translated to an appropriate view. In order to represent the right data in the appropriate form, the needed view is dependent on the domain of the query. The domain knowledge needs to be translated to an automatically usable model.
Solution — Queries sent by the user interface should be translated somehow in

order to know what should be send back. The QMB only creates views of the data but does not know how to relate this to the user interface. The use of a query handler can solve this problem by implementing a component able of receiving all queries and checking the query store for views created by the QMB (Torkel, 2010). Figure 5.6 shows a combination between the QMB pattern (section 5.5.5) and the

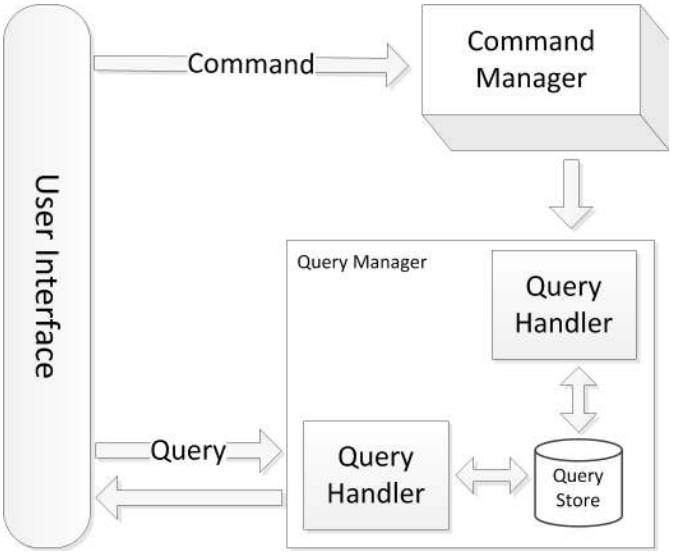


FIGURE 5.6: QMB and Query Handler Sub Pattern

query handler pattern. The concept of a query store is introduced to store queries build by the QMB. This store is not obligatory, but can improve the response time of the system.

Consequences — Queries are now translated to views, usable for representation to tenants through the user interface.

5.5.7 Snapshotting

Problem — States only occur in aggregate roots, but recovering the state of an aggregate root after a system crash is intensive.

Solution — It is common practice in the CQRS pattern to only store changes (events) and no states. This is because states can always be determined based on all the changes happened in the system so far. Rerunning al events will bring the system back in its last state after a possible system crash. States only occur in aggregate roots (see section 5.5.3), but recovering the state of an aggregate root after a system crash can be quite intensive, since aggregate roots often stay active in the system for a long time. A solution to this problem is the use of snapshotting.

In the snapshotting pattern, the state of the aggregate root is stored together with the events every n^{th} event. The exact value of n depends on the processing load storing and monitoring the state of the aggregate root gives. When the system crashes, the latest stored state is recovered, and only the events happened after this state storage have to be rerun. The snapshotting pattern is often used in combination with the memento pattern (Gamma, Helm, Johnson, and Vlissides, 1995) that provides the ability to restore objects to their previous state.
Consequences — System recovery is faster and more reliable.

5.6 Variability Influences

Applying the CQRS pattern in a software product does not immediately influence the level of variability of a software product. Applying CQRS in combination with sub patterns identified in this case study however does have a positive effect on the variability level of a software product. On a functional level, it becomes possible to comply to specific customer groups or branches of industry having their own specific requirements.

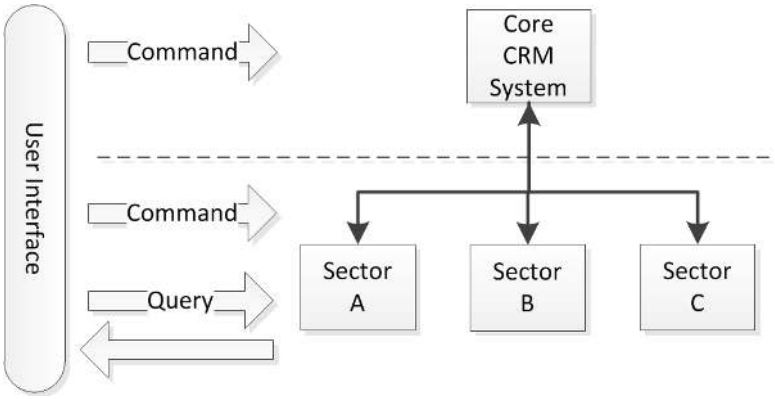


FIGURE 5.7: Example of variability due to CQRS

Figure 5.7 shows how applying the *Event Sourcing* and *Event Store* patterns (section 5.5.1) can help in offering specific functionality to different industrial sectors. The example is an adapted version of a design observed at ERPComp. The figure shows three different (A, B and C) sectors, but this can differ per implementation. In the system, one core system is created containing the functionality that is shared by all sub systems. For example, a CRM system, having specific sub systems for sectors like retail, furniture and bakeries. All domains share names and addresses for customers, so commands related to this would be performed by *Aggregate Roots* in the core CRM system. All branches would listen to events

broadcasted by the core CRM system and build query models based on events that are relevant for them. Operations on attributes or entities that only exist on one of the sub systems (for example a membership card number for retail) will only be processed within the specific sub system. The core system should only receive commands and does not have to be able to process queries, since the sector specific sub systems handle the representation of specific data. By identifying the different requirements of customers and grouping them in different sub systems, the level of variability possible within a software product will be high.

The possibility of running *Query Model Builders* (section 5.5.5) at the client side, also opens possibilities for customer specific requirements that are not shared with other customers. Custom *QMBs* and *Query Handlers* can be developed and implemented at customers, allowing them to perform the specific task needed for their business process. The customer specific listeners listen to events broadcasted and can react in a way specific for the wishes of one customer. The location of deployment does not play a role, making it possible to run QMBs at the customer, but also at third parties. Overall, as observed within ERPComp, the CQRS pattern enables software vendors to create a software product better capable of complying to all sort of different customer requirements and by this achieve a high level of variability. This is primarily caused by the possibility to distribute events to specific event listeners and the ability to handle those events in a way that can be customer or customer type specific.

5.7 Discussion and Future Research

Software products designed according to CQRS and sub patterns identified in this research can profit from an optimal configuration of data stores in such a way that it is geared towards a specific task (i.e. storing or reading data). By this, the CAP theorem can be less of a problem than it would have been if one data store had to do all tasks. The distributed asynchronous way in which events are handled is primarily useful for business software product having a high concurrent load or a high need for variability. Business products, as analyzed within our case study, have both of the characteristics described above and will benefit from applying the CQRS pattern, including identified sub patterns, in terms of scalability, performance during load peaks and the level of variability.

The sub patterns reported on in the paper are all based on the patterns applied within the case company, cross checked with patterns currently described in CQRS

related literature. The selection of patterns described is not a complete set of patterns related to CQRS. More and different patterns exist, but the patterns identified in this research are those used within ERPComp. Furthermore, extensive research at additional case companies can extend the collection of CQRS sub patterns and create an even more complete overview of CQRS related patterns.

Future research should also make clear when architects should choose certain patterns and how these different sub patterns work together to achieve some common goal. The characteristics of all patterns should be more extensively evaluated by domain expert to create a complete catalogue of all CQRS related patterns.

5.8 Conclusion

As the case study and variability example illustrates, the CQRS pattern can help in achieving a high level of variability in a software product. Different sub patterns are identified related to CQRS to solve specific problems within a CQRS based architecture design. This paper helps software architects by explaining the different sub patterns and showing how they can influence the variability off a software product.

We showed an implementation of the CQRS pattern, including seven sub patterns that are observed at ERPComp. This example shows how implementing a CQRS based architecture instead of a multi-tier architecture can help in creating a software product capable of serving thousands of customers with variable product requirements. All identified sub patterns can be implemented together or individually to create, but none of them are obligatory for implementing the CQRS pattern. Some sub patterns, like the Event Sourcing pattern and the use of distributed query model builders, can contribute directly to the variability of a software product in a significant way. Other sub patterns however have a supporting role for the architecture, dealing with scalability, performance or consistency of the system. There is no perfect combination of sub patterns when it comes to CQRS since everything specific situation differs, but the pattern descriptions in this paper help in making a weighed decision for software architects.

Part II

Selecting Patterns in Systems Design

Chapter 6

Multi-Tenant Architecture Assessment

Abstract

Software architects struggle to choose an adequate architectural style for multi-tenant software systems. Bad choices result in poor performance, low scalability, limited flexibility, and obstruct software evolution. We present the Multi-tenant Architecture Assessment Model (MAAM) that supports architects in choosing the most suitable architectural pattern, among a set of 12 Multi-Tenant Architecture (MTA) patterns and using 17 assessment criteria. Both patterns and criteria were evaluated by domain experts. Five architecture assessment rules of thumb are presented in the paper, aimed at making fast and efficient design decisions. MAAM provides architects with an effective method for selecting the applicable multi-tenant architecture pattern, saving them effort, time, and mitigating the effects of making wrong decisions.

This work has been published as a short paper named *Multi-tenant Architecture Comparison* to the 8th European Conference on Software Architecture (ECSA2014). It is co-authored by Michiel Pors, Slinger Jansen and Sjaak Brinkkemper

6.1 Introduction

As a consequence of the current shift of on-premises software to the cloud (D'souza, Kabbedijk, Seo, Jansen, and Brinkkemper, 2012), software architects find themselves facing numerous new challenges related to the adequacy of architectures for cloud software. A commonly used technique in architecting for Software-as-a-Service (SaaS) is the use of the concept of multi-tenancy, which is defined for this research as *"a property of a system where multiple customers, so-called tenants, have the possibility to configure the system; it allows them to transparently share the system's services, applications, databases, or hardware resources, with the aim of lowering costs"*(See chapter 2 for more details).

Multi-tenancy can bring about many benefits. By serving the software service from a centrally hosted location, clients are relieved from the responsibility of purchasing and maintaining expensive in-house servers. The total cost of ownership decreases, giving the SaaS provider access to new potential customers that previously could not afford the expenses (Chong and Carraro, 2006). In addition, the utilization rate of hardware in a multi-tenant environment is higher than in a single-tenant environment (Sääksjärvi, Lassila, and Nordström, 2005). Furthermore, when multiple customers share application and data instances, the total number of running instances will be lower than in a single-tenant environment, catering the same number of customers. A low number of instances is beneficial for maintenance (Kwok, Nguyen, and Lam, 2008) and is beneficial for application development (Bezemer, Zaidman, Platzbeecker, Hurkmans, and Hart, 2010).

However, multiple barriers withhold service providers from massively switching to multi-tenant environments. The challenges for multi-tenancy adoption include performance (Lin, Sun, Zhao, and Han, 2009), scalability, security (Guo, Sun, Huang, Wang, and Gao, 2007), and the re-engineering of current software applications (Tsai, Ruan, Sahu, Shaikh, and Shin, 2007). Selecting the appropriate multi-tenant architecture is a complex problem due to the existence of numerous alternative architectural patterns. Benefits and barriers of multi-tenancy are identified and described in literature, but the aspect of choosing an appropriate multi-tenant architecture based on software vendors' preferences has received little attention in literature.

Finding the most suitable multi-tenant architecture is crucial; it expresses a fundamental structural organization schema for a provider's software system. However, choosing the appropriate architecture is a wicked problem (Esfahani, Razavi, and

Malek, 2012). Accounting for all the challenges and benefits complicates the decision process considerably (Kazman, Asundi, and Klein, 2001). Previous studies in multi-tenant architectural decision making exist (Esfahani, Malek, and Razavi, 2013) and often focus on a select set of quality attributes (Koziolek, 2011; Momm and Krebs, 2011) while assessing architectural decision making, or focus primarily on quantitative data from test deployments and specific implementations (Wang, Guo, Gao, Sun, Zhang, and An, 2008). The consequences of applying a specific pattern are dependent on the implementation. Because of this, assessment of the architecture by experts based on their experiences is needed, leading to a high-level analysis of the architecture. The Multi-tenant Architecture Assessment Method (MAAM) aims at filling this gap by providing a concise and flexible method for multi-tenant architecture decision making.

This paper presents the MAAM in section 6.3, based on the mixed-method research approach used within this study (section 6.2). The twelve different Multi-Tenant Architectures (MTAs) are shown in section 6.4, together with the list of MTA assessment decision criteria in section 6.5. The MTA decision matrix is explained in section 6.6, together with a collection of rules of thumb, supporting time-efficient design decisions. We conclude with a discussion of MAAM, together with threats to validity present and future work in section 6.7, focussing on the importance of evaluating more effective methods in architectural decision making.

6.2 Research Approach

The main research question of this research is formulated as follows:

RQ. *How can a SaaS provider be supported in the decision process of choosing an applicable multi-tenant architecture pattern?*

Three sub questions are answered in order to develop a decision model that answers the main research question. The decision model consists of three fundamental elements, which need to be identified. The first element is a set of multi-tenant architectures to choose from. Hence, the first sub question is defined as follows:

SQ1. *What distinctive layers in multi-tenant architectures can be defined?*

Using a Structured Literature Research (SLR) approach, the distinctive layers in multi-tenant architectures are identified to answer SQ1. Instead of searching

directly for multi-tenant architectures, different layers on which multi-tenancy can be applied are first identified. Based on the different multi-tenancy layers, generic multi-tenant architectures patterns are composed. The list of candidate architecture pattern is validated by ten software architects to ensure the list is complete and concise. The expert validation is not only essential for checking the correctness of the list, but also to make sure the identified architectures reflect *relevant* and *implementable* architectures. Additionally, the construct and external validity of the list is enhanced by performing the validation.

SQ2. *What are the relevant decision criteria for choosing an appropriate multi-tenant architecture pattern?*

SQ2 aims at identifying the different decision criteria, or architecturally significant requirements, related to multi-tenant architectures. The decision criteria are aimed to be attributes, distinguishing in the decision between different multi-tenant architecture patterns. Similar to the identification of the MTAs, a structured literature research is carried out to identify the list of criteria. The identification process results in a set of candidate criteria, which is analyzed in order to merge similar and delete unimportant attributes. Consequently, the completeness and conciseness of the reduced list is validated in an expert evaluation, consisting of the same ten software architects used to answer SQ1.

SQ3. *How do the different multi-tenant architecture patterns perform on the decision criteria?*

In order to answer SQ3, an evaluation is performed in which all MTAs are evaluated on the identified decision characteristics. The evaluation takes place by surveying 16 software architects from three different enterprise software companies, all currently offering online enterprise software on a large scale. All experts are asked to fill in a survey, querying about the effect of applying the MTAs in a software product, on the decision characteristics. By using a structured survey on 16 experts from tree different enterprise software companies, the validity of the evaluation is ensured.

6.2.1 Structured Literature Research

A structured literature survey is conducted using an explicit search strategy as prescribed by Kitchenham (Kitchenham and Charters, 2007). As there were no

systematic literature reviews on the topic of multi-tenant architecture evaluation, we conducted a traditional search process in the major digital libraries in the domain of software engineering and architecture:

1. ACM Portal; 2. IEEE Xplore Digital Library; 3. ScienceDirect; 4. SpringerLink; 5. Scopus

Using the following search string:

> ABSTRACT:((tenan* OR multitenan*) AND (software or service OR application or saas)) AND KEYWORDS:(tenan* OR multitenan*)

An asterisk is used as a wild-card and represents variations of the corresponding word, e.g., tenan* represents tenant and tenancy. The search string is constructed by linking the two or lists using the boolean AND. For more details on the search strategy and construction of trail searches, please see our previous technical report (Pors, Blom, Kabbedijk, and Jansen, 2013).

Study selection criteria assess the relevance of the literature found in the first step. The selection criteria are piloted on a subset of primary studies. The initial electronic search results in a large number of irrelevant papers, and using these criteria a smaller, more relevant list of literature can be created. The following criteria are used:

Inclusion Criteria

1. any article focusing on the topic of multi-tenancy in a hardware or software environment.
2. any article either describing multi-tenancy levels, decision criteria, or both.
3. any article that is cited by other literature in the description of multi-tenancy levels.

Exclusion Criteria

1. articles that do not appear in scientific papers or conference proceedings.
2. articles already obtained by other digital libraries.

3. articles written in a different language than English.

4. articles of which no full copy can be obtained.

After identification through the SLR, the MTAs and decision criteria are consolidated and evaluated by domain experts to ensure their validity.

6.2.2 Expert Validation

The evaluation of the multi-tenant architectures and decision criteria is conducted using a questionnaire. Using this survey ten experts are asked for their opinions on (1) the structured multi-tenant architectures and (2) the composed set of decision criteria. The experiences of the experts range from 2 years to 27 years, with over 14 years average experience in software architecture. The ten software architects work for Dutch organizations with large cloud deployments in the public service sector.

Inclusion of the multi-tenant architectures and decision criteria (as presented in section 6.5) in the assessment model depends on the median of the evaluation scores given by the experts. The median describes a numerical value separating the higher half of a list of numbers from the lower half. If the list has an even number of items, the median is defined as the mean of the two middle values. The median is a more robust measure of central tendency in the presence of outlier values than is the mean (Stevens, 1946). All answers in the questionnaire use a 7-point Likert scale.

If the median MTA evaluation score is equal to or greater than 3, it is included in the assessment model. This threshold is chosen, because the third Likert item is semantically described as slightly feasible, which means the experts rate the item as feasible enough to be included in the assessment model. The decision criteria are evaluated on two requirements; therefore, inclusion of a decision criterion depends on two medians. If they are both equal to, or greater than 3, the corresponding criterion is included in the assessment model, because the third Likert item, described as a slight distinction or deciding factor, is considered sufficient.

6.3 Multi-Tenant Architecture Assessment Model

The Multi-tenant Architecture Assessment Model, as depicted in Figure 6.1, is proposed to support software architects and other decision makers in the assessment of appropriate multi-tenant architectures. MAAM uses a different approach than well-know architecture evaluation methods like ATAM (Kazman, Klein, and Clements, 2000), since it focusses exclusively on multi-tenant architectures and the decision criteria related to MTAs. The model consists of three phases, in which several steps are carried out using three artifacts depicted on the "uses" level.

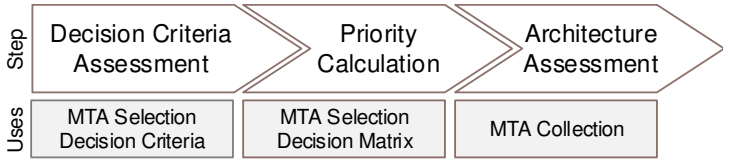


FIGURE 6.1: The Multi-tenant Architecture Selection Model: Steps and Artefacts

Decision Criteria Assessment - A SaaS provider initiates the decision process with the assessment phase. This phase comprises of assessing the criteria set on completeness and minimum size. The artifact used for this phase is the list of MTA assessment decision criteria, which can be found in section 6.5. First, a SaaS provider needs to assess the completeness of the criteria set. This means he has to determine if each factor influencing the decision problem for that specific SaaS provider is covered by a criterion. If this is not the case, the SaaS provider can opt to add criteria. In case no criteria are added to the list, just the minimum size property should be re-evaluated. The model aims to provide a complete set of criteria while keeping flexibility if a SaaS provider's domain requires so.

Priority Calculation - In this phase, the calculation of relative priorities of the criteria takes place. Weights need to be assigned to each criterion. This can be done by using an absolute measurement in which each criterion directly receives a value lying between a predetermined range, representing the importance of that criterion. Or, using the relative measurement in which criteria on equal level in the hierarchy are compared with each other on relative importance with respect to their common parent. Then, together with the MTA assessment decision matrix in Table 6.3 global priorities can be calculated for each multi-tenant architecture.

Architecture Assessment - In case multiple architectures receive high priorities lying close to one another, a deeper analysis between these architectures needs to be conducted. This analysis should make use of more qualitative data and quality trade-offs. When there is a single preferred multi-tenant architecture, decision

Multi-tenancy level	*N*
Application Instance	16
Database Server	16
Database	15
Operating System	15
Hardware	14
Schema	14
Middleware	12
Virtual Machine	9
Application Server	4

TABLE 6.1: Multi-Tenancy levels identified in literature

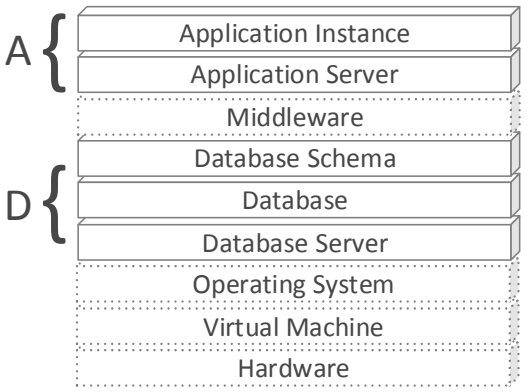


FIGURE 6.2: Multi-tenancy computing stack. *'A' and 'D' relatively indicate the Application and Data related layer sets*

makers should validate if this architecture in fact meets requirements and achieves the expected goals.

6.4 Multi-tenant Architectures

The levels at which multi-tenancy can be applied, resulting from the literature study, are shown in Table 6.1. All levels are listed together with the frequency of occurrence (N) in literature. The different levels are depicted as layers in a stack with decreasing granularity from top to bottom in Figure 6.2. The granularity aspect translates to a sharing versus isolation continuum, where the lowest layer has the lowest level of sharing with the highest level of isolation. For the highest layer, it is vice versa. When multi-tenancy is applied at a specific level, the levels below that level are shared among tenants as well, but isolation occurs at the levels above, i.e. for each tenant a dedicated instance is running. This applies to the application and data layer independently. For example, when multi-tenancy is applied at the application server level, the application server, virtual machine instance and hardware are shared among tenants. Isolation occurs at the levels above the application server, so each tenant receives a dedicated application instance, but multi-tenancy in the data layer can be applied on a different level.

The final two levels of the stack in the data layer are the *database* and *schema level*. These two levels were first described by Chong et al. Chong, Carraro, and Wolter (2006). When tenants are consolidated in a single database, each tenant

operates its own set of tables. In schema-level multi-tenancy, isolation occurs at table row level.

In cloud computing, an *infrastructure provider* manages and controls the infrastructure consisting of processing, storage, networks and other fundamental computing resources (Mell and Grance, 2011). For a *service provider*, which develops the application and is the primary stakeholder in this research, the aspect of multi-tenancy in these lower levels is little importance. It has no influence on the architectural design decision of the software product. The number of servers, instances and databases is far more relevant for a service provider (Dillon, Wu, and Chang, 2010). For this reason, the hardware, virtual machine, operating system and middleware levels are not considered in structuring different types of multi-tenant architectures in this research.

The *application* related layer set (A) and the *data* related layer set (D) are stacks commonly used in enterprise architecture in order to separate concerns (Fowler, 2003). Within this research the application layers and data layers are identified as separate *layer sets*, each containing different sub-layers, as can be seen in Figure 6.2.

Consequently, three tenancy levels, indicated by a two-letter abbreviation, are identified in the *Application* related layer set (**A**). The different levels result from identifying ascending levels of sharing among all layers on the set:

1. **AD** - A Dedicated Application server is running for each tenant, and therefore, each tenant receives a dedicated application instance.
2. **AS** - A single Application Server is running for multiple tenants, and each tenant receives a dedicated application instance.
3. **AI** - A single application server is running for multiple tenants, and a single Application Instance is running for multiple tenants.

The first level corresponds to multi-tenancy enabled at the hardware or virtual machine level. The second level is equal to application server multi-tenancy. The third level is the same as multi-tenancy enabled at the application instance level. In the *Data* related layer set (**D**) a service provider can select one the following four tenancy levels:

1. **DD** - A Dedicated Database server is running for each tenant, and therefore, each tenant receives a dedicated database.

2. **DS** - A single Database Server is running for multiple tenants, and each tenant receives a dedicated database.

3. **DB** - A single DataBase server is running for multiple tenants, data from multiple tenants is stored in a single database, but each tenant receives a dedicated set of tables.

4. **DC** - A single database server is running for multiple tenants, data from multiple tenants is stored in a single database and a single set of tables, sharing the same Database sChema.

The first level is equal to multi-tenancy applied at the hardware or virtual machine level. The second one corresponds to database server multi-tenancy. The third alternative is the same as multi-tenancy applied to the database, and the final one is equal to database schema multi-tenancy.

From these options in both the application and data layer, the set of multi-tenant architectures (MTAs) are constructed. Based on the tenancy levels within the layers, the number of possible architectures is twelve. Because all MTAs prescribe a specific tenancy level in set A and D, each architecture is defined as a tuple:

$$MTA = \langle \{AD, AS, AI\}, \{DD, DS, DB, DC\} \rangle \tag{6.1}$$

Each of the twelve MTAs can be seen as an architectural pattern in which tenants (Tenant A, B and C in the example MTAs) communicate with a software application consisting of an application layer and a data layer as shown in Figures 6.3 and 6.4. Two MTAs out of twelve are shown here; for a complete overview of all MTAs, please see (Pors, Blom, Kabbedijk, and Jansen, 2013).

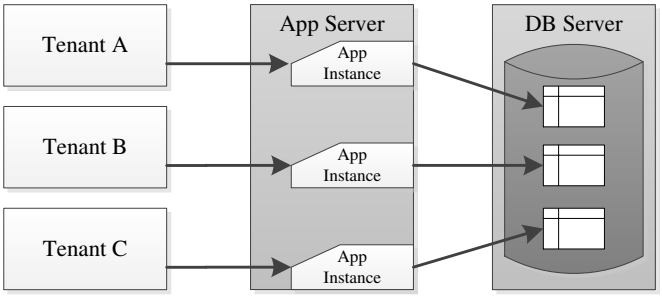


FIGURE 6.3: MTA$\langle AS, DB \rangle$ - Shared Application Server & Shared Database

In Figure 6.3 and 6.4 the application layer is represented as a set of application servers running one or multiple application instances. The data layer is displayed

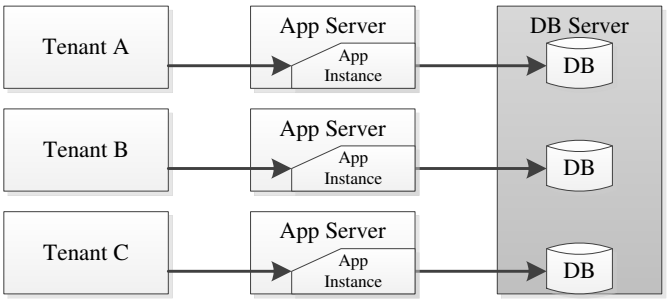


FIGURE 6.4: MTA$\langle AD, DS \rangle$ - Dedicated Application Server & Shared Database Server

as a set of database servers, running one or more databases, in which one or multiple database schema's exist. If one of these entities is shared among the tenants, its color is gray. If it is dedicated to only one tenant, its colored white. For the sake of simplicity, only three tenants are displayed in the architectures. A service provider can offer his software application to more than three tenants, the patterns merely present possible arrangements of shared resources.

6.4.1 Expert Validation

Experts are asked to rate the feasibility of the different MTAs, with 1 being the lowest and 7 being the highest feasibility score.

$\langle AD, DD \rangle$ receives a high degree of feasibility, seven experts defined it as at least a very strongly feasible architecture. The opinions are more divided $\langle AS, DD \rangle$, but the majority agrees it represents at least a moderately feasible architecture. $\langle AI, DD \rangle$ proves to be the lowest feasible architecture with an aggregate value between slightly and moderately. Three experts define $\langle AD, DS \rangle$ as slightly feasible, yet five experts define it is at least very strongly feasible. On $\langle AS, DS \rangle$ no real consensus is reached as well, but the collective is stated as having a value between moderately and strongly feasible.

Opinions on $\langle AI, DS \rangle$ are even more divided as each Likert item is checked. Its shared value is moderately feasible. The extent of feasibility on $\langle AD, DB \rangle$ is a bit higher with a value between moderately and strongly. Half of the experts define $\langle AS, DB \rangle$ as at least very strongly feasible. On the extent of feasibility on $\langle AI, DB \rangle$, the judgments can be divided into two equally large groups. One stating it is slightly feasible at best, and the other stating at least strongly feasible. The joint value however is moderately feasible. Six out of ten experts define $\langle AD, DC \rangle$ as a strongly feasible architecture, equal to its aggregate value. $\langle AS, DC \rangle$ receives a strong degree of feasibility and on $\langle AI, DC \rangle$ experts concur on a very strong

degree of feasibility. For all MTAs, $\mu_{1/2} \geq 3$ applies and therefore all architectures are included in the MAAM. The median is not calculated to identify differences between the architectures, but to check per architecture individually how they score in terms of feasibility.

6.5 MTA Assessment Criteria

106 criteria are initially identified from literature. The initial set potentially includes irrelevant and redundant attributes, meaning it does not yet adhere to the minimal and non-redundant principle. The list of criteria is, therefore, reduced in several evaluation steps. Figure 6.5 illustrates the process how the initial list of criteria is condensed to the final list of evaluation criteria. The left side of the figure shows the activities in the SLR selection process, while the right side of the figure shows the deliverables from these activities.

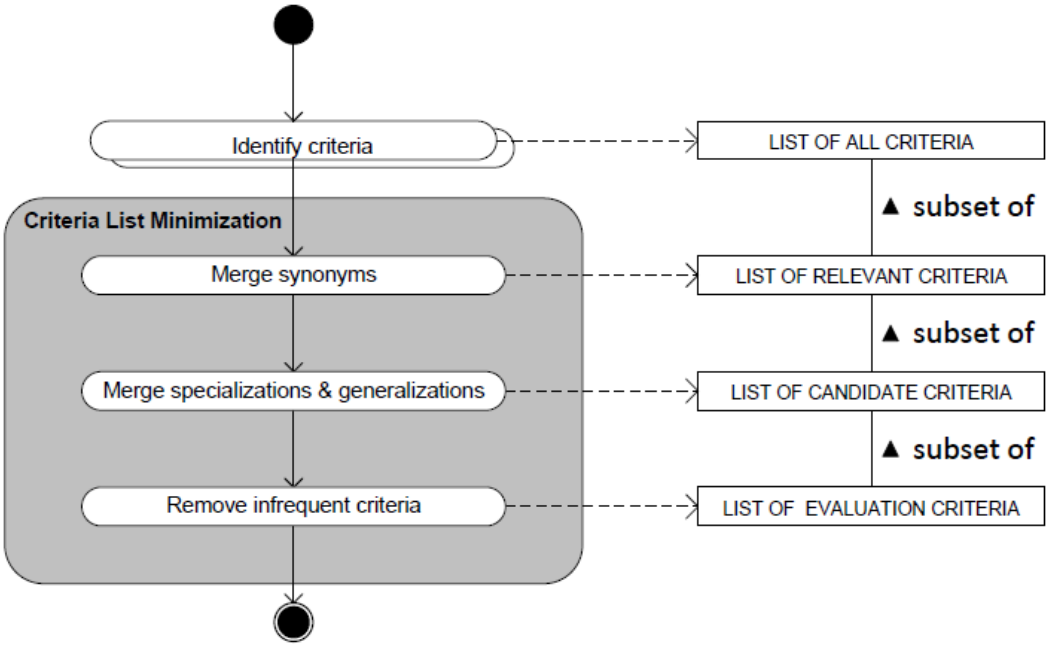


FIGURE 6.5: Process Deliverable Diagram of MTA Assessment Criteria Identification

The first step in MTA assessment is identifying and selecting the relevant decision criteria in accordance with the architects. Consequently, the initial set of decision criteria is reduced by merging synonyms. After this step, criteria representing specializations and generalizations of other criteria are combined. The final step to reduce the list is by deleting infrequent criteria, i.e. attributes that are identified less than five times in literature. The list of evaluation criteria resulting from the

last step is used as input for the expert evaluation discussed in section 6.5.1. The majority of merging combinations is straightforward, but some decisions need additional investigation before they can be merged. The most significant merges of criteria will be shortly discussed below. For a complete discussion of all criteria, please see Pors, Blom, Kabbedijk, and Jansen (2013).

Scalability is frequently identified in multi-tenancy literature as an important advantage and area of interest. Bondi defines scalability as a desirable ability of a system, network, or process to accommodate an increasing amount of elements and process this accompanying extra volume of work in a capable manner (Bondi, 2000). Additional workload is required when the service is offered to extra tenants or users. As a result, scalability is related to the number of tenants and users an architecture can support. For that reason, scalability is merged with both the number of tenants and the number of users.

In the elaborate Computer Science Handbook, fault tolerance is described as "the total number of failed elements that can be present without causing output errors" (Tucker, 2004, p. 649). The reliability of a system is defined as "the probability that the system will produce correct outputs up to time t, provided it was producing correct outputs to start with" (Tucker, 2004, p. 646). The availability of a system is defined as "the probability that the system is operational at time *t*" (Tucker, 2004, p. 646). Because a reliable system will be operational and produce correct outputs, even when there are failed elements, *fault tolerance* and *reliability* are both merged with availability.

The relationship between access control and authorization and authentication is extensively discussed by Sandhu and Samarati (1994). The authors define *access control* as "to limit the actions or operations that a legitimate user of a computer system can perform. Access control constrains what a user can do directly, as well what programs executing on behalf of the users are allowed to do. In this way, access control seeks to prevent activity that could lead to a breach of security"(Sandhu and Samarati, 1994, p. 1). *Authentication* is concerned with correctly establishing the identity of the user, while authorization relates to determining if the user attempting to do an operation is authorized to perform that operation. The effectiveness of access control depends on proper authentication and correct authorization, causing both *access control* and *authorization* to be merged with authentication.

The result of these steps is the list of evaluation criteria, as illustrated in Table 6.2. The column after the decision criterion shows how many times (N) that criterion

is identified from the list of selected literature. The criteria listed in Table 6.2 are evaluated by domain experts in Section 6.5.1.

Criterion	*N*	Criterion	*N*
Variability	65	Authenticity	12
Number of Tenants	60	Confidentiality	11
Maintainability	45	Deployment Time	9
Number of End-Users	44	Flexibility	9
Resource Utilization	42	Throughput	8
Software Complexity	32	Monitoring	7
Integrity	23	Diverse SLA	5
Time Behaviour	21	Portability	5
Availability	16		

TABLE 6.2: Selection Criteria identified in Literature

The attribute operating cost covers a broad range of expenses, e.g. business overhead costs and equipment operating costs. All attributes in Table 6.2 can be associated with certain types of costs. The operating cost attribute encompasses most costs associated with these other attributes. For this reason, the operating cost attribute will not be included in the assessment model. Some of the criteria identified are equal to or synonymous of the quality characteristics of software products and computer systems defined in ISO/IEC 25010 (ISO/IEC, 2011), which are used to define the quality of software and computer systems. The ISO/IEC 25010 standard contains a different list of attributes than the list identified in this research. This discrepancy is caused by the focus on multi-tenancy, causing some ISO attributes to be obsolete or lacking.

6.5.1 Expert Evaluation

Experts are asked to rate to what extent the specific criterion is influenced by an MTA selection choice, leading to a **Distinction Value**. Experts were also asked to assess the effect of the criterion on the MTA assessment process, which is defined as the **Deciding Factor**.

According to the experts, *Time Behavior* has a high distinction and deciding factor. *Resource Utilization* holds a high distinction, but the ratings on the deciding factor are more scattered. Still, more than half of the experts define the criterion as having at least a strong deciding factor. The distinction ratings of *Throughput* are spread, but the current value of five indicates is has a strong distinction. The aggregate value equates to moderate. On the deciding factor of Throughput, no

Likert item is checked more than twice. Here too, the aggregate value equals moderate. There is a better consensus on *Number of Tenants* with a high distinction factor and a strong deciding factor. Lesser agreement exists for the distinction on *Number of End-users*, but seven experts define it at least as strongly distinct.

For *Availability* six experts state it has at least a strong distinction and seven experts stating it has at least a strong deciding factor. The criterion of *Confidentiality* shows similar scores, and there is consensuses on both factors. There is less consensus on both factors of the *Integrity* criterion, yet six experts find it has at least a strong distinction and deciding factor. *Authenticity* receives the lowest aggregate value on distinction. Six experts state a slight distinction among the architectures and five experts state it has a strong deciding factor.

On *Portability* there is a good consensus, but the degree of distinction and deciding factor is only moderate. Eight experts state *Variability* distinguishes to a degree between moderately and very strong. Seven experts state its deciding factor lies on moderate or strong. For *Diverse SLA* there are again eight experts stating it distinguishes moderately to very strong. A majority answered the deciding factor with a moderate extent. For *Software Complexity* there are seven experts defining it as at least distinction strongly and six experts defining it as an at least strong deciding factor. Finally, there exists a high consensus for *Monitoring* where nine experts agree it distinguishes strong or very strong, and seven experts agree it has a strong or very strong deciding factor.

Both medians for each decision criterion are equal to greater than 3 and therefore each decision criterion is included in the final assessment model.

6.6 MTA Decision Matrix

The MAAM offers software architects a method to make an informed and balanced decision on the MTAs to consider implementing for their software product. The MTA Decision Matrix in Table 6.3 enables software architect to select the most suitable multi-tenant architecture, based on expected usage performance. The matrix shows the average rating of the 16 experts on the effect of an MTA on a specific decision criterion. A value of 1 indicates a detrimental effect, while a score of 5 indicates a positive effect. The last column of the matrix shows the σ^2-value, indicating how much the criterion is affected by the choice for a specific MTA. Using the matrix, architects can get an overview of the consequences of all

Decision Criterion	$\langle AD, DD \rangle$	$\langle AS, DD \rangle$	$\langle AI, DD \rangle$	$\langle AD, DS \rangle$	$\langle AS, DS \rangle$	$\langle AI, DS \rangle$	$\langle AD, DB \rangle$	$\langle AS, DB \rangle$	$\langle AI, DB \rangle$	$\langle AD, DC \rangle$	$\langle AS, DC \rangle$	$\langle AI, DC \rangle$	Dist. Factor (σ^2)
Time Behavior	5.0	4.0	4.0	4.0	3.0	3.0	4.0	3.0	3.0	3.5	3.0	2.5	0.5
Resource Utilization	2.0	2.5	3.0	2.5	3.0	3.0	3.0	3.0	4.0	3.0	3.0	4.5	0.4
Throughput	4.5	3.0	3.0	4.0	3.0	3.0	3.5	3.5	3.0	3.0	3.0	3.0	0.2
Number of Tenants	1.0	3.0	3.0	3.0	3.5	4.0	3.0	4.0	4.0	4.0	4.0	5.0	1.0
Number of End-Users	2.5	3.5	3.0	3.0	3.5	3.5	3.5	3.5	4.0	3.5	4.0	4.5	0.3
Availability	4.0	3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.5	3.0	3.0	3.0	0.1
Recoverability	5.0	4.5	4.5	4.0	4.0	4.0	3.0	3.0	3.0	2.0	2.0	2.0	1.1
Confidentiality	5.0	4.5	4.0	4.0	4.0	4.0	3.5	3.0	3.0	2.0	2.0	2.5	0.9
Integrity	4.5	4.0	3.0	4.0	3.5	3.0	3.5	3.0	3.0	3.0	2.5	2.0	0.5
Authenticity	4.5	3.5	3.0	3.5	3.0	3.0	4.0	3.5	3.0	3.0	3.0	2.5	0.3
Maintainability	1.5	2.5	3.0	2.5	3.0	3.5	2.5	4.0	4.5	3.0	4.0	5.0	1.0
Portability	5.0	5.0	4.5	4.5	4.5	4.5	4.0	4.0	4.0	3.0	3.0	2.5	0.7
Deployment Time	1.5	3.0	3.0	2.5	3.5	4.0	3.0	4.0	4.0	3.0	4.0	5.0	0.8
Variability	5.0	4.0	2.5	5.0	4.0	2.0	4.5	3.5	2.0	2.5	2.0	1.0	1.8
Diverse SLA	4.5	4.0	3.0	4.0	3.5	2.5	4.0	3.0	3.0	3.0	2.5	2.0	0.6
Software Complexity	5.0	4.5	4.0	4.5	4.5	3.5	4.0	4.0	3.0	2.5	2.5	2.0	0.9
Monitoring	1.0	2.0	3.0	2.5	3.0	3.0	3.0	4.0	4.0	3.5	4.0	5.0	1.1

TABLE 6.3: Multi-Tenant Architecture Decision Matrix (In color) - $n = 16$

different MTA patterns and assess the weight of the consequences for their specific situation. Based on the consequences and the weights, architects can select a subset of patterns to evaluate in more depth, using, for example, the Software Pattern Evaluation Method (SPEM) presented in Chapter 8. To help in selecting a subset for future analysis, this section presents some Rules of Thumb (RT) derived from the decision matrix and are helpful in giving decision makers a quick overview of the most important consequences of an MTA assessment.

RT1. Focus on the database dimension — The effect of different MTAs on decision criteria is largest on the database dimension. The MTA Decision Matrix shows the effect of database related decisions is higher than application related decisions. Choosing between a set of MTAs, focus on database related decisions first, and application related decisions after.

RT2. Sharing database tables enables serving of many tenants but harms robustness — Selecting an MTA in which the database schema is shared (i.e. $\langle A?, DC \rangle$[1]) is beneficial if the software product serves many tenants and end-users. The product is easy to maintain and monitor, and deployment time

[1] '?' is used as a single character wild card.

is minimal. The recoverability of the system, on the other hand, is greatly compromised. It is difficult to implement variability and tenant data may be at risk of unintentional sharing. Based on this trade-off, SaaS providers should select $\langle A?, DC \rangle$ when designing a large scale software product with limited variability requirements.

RT3. Sharing application instances helps maintainability and performance, but harms variability — Choosing an MTA, decision makers can decide to share the application instance among tenants (i.e. $\langle AI, D? \rangle$). Doing so causes the maintainability and ease of monitoring to increase. Also, the resource utilization is better and the deployment time low. The variability of the software product, however, is lower and more difficult to implement. Because of this, SaaS should choose $\langle AI, D? \rangle$ when maintainability and performance efficiency are important.

RT4. Ease of implementing variability differs greatly per MTA — Out of all decision criteria, variability has the highest distinction factor. This means the variability of a software product is for a significant part determined by the implemented MTA. Choosing an MTA with a low tenancy level (i.e. $\langle AD, DD \rangle$, variability is relatively easy to achieve. Selecting an MTA with a high tenancy level however (i.e. $\langle AI, DC \rangle$), causes large problems implementing variability among all tenant instances.

RT5. Dedicated servers improve performance and variability, but hamper scalability — When choosing an MTA with dedicated servers (i.e. $\langle AD, DD \rangle$) the time behavior, recoverability, variability and confidentiality are expected to be good, and software complexity low. The downside to this approach is the low scalability of the system; when the number of tenants increases, dedicated servers become hard to maintain, and hardware costs will rise. Choose $\langle AD, DD \rangle$ for software products with a small user base that need to have high performance and a high level of flexibility. Typically large enterprise applications fall in this category.

The rules of thumb listed in this section do not aim for completeness, but rather give software architects and decision makers a collection of rules to guide their architecture selection.

To illustrate, we take the fictional company FictComp, which is a small software company who currently produce an on-premises software product, but expect to grow significantly in the near future. They want their online product to be able to support a *large number of tenants* and *end-users* but will not offer very *complex*

or *variable* functionality. As a starting point, FictComp assesses the five rules of thumb. RT2 indicates that sharing database tables is beneficial for FictComp and enables them to serve a large number of tenants. Because variability plays a minor role, FictComp decides to also share the application instance among tenant as described in RT3. These decisions lead FictComp to $\langle AI, DC \rangle$ and gives them a starting point to further analyze the consequences of this specific MTA choice in 6.3. Using the matrix, they can now adjust or change their choice accordingly.

6.7 Discussion and Conclusion

The MAAM, and the identification of the 12 different multi-tenant architecture patterns, along with a list of assessment criteria and rules of thumb, supports SaaS providers in providing a concise and versatile method for multi-tenant architecture assessment. In case specific assessment criteria or MTAs are irrelevant to a software architect, those elements can be easily removed from the analysis, simplifying the selection of a suitable architecture. If an architect feels important decision criteria are missing from the assessment model, extra decision criteria can be added in the analysis. However, performance values of the MTAs on these criteria are provided in this research.

We identify the following threats to validity to this study: 1. The set of twelve multi-tenant architecture patterns does not take into account possible hybrid patterns in which different solutions are used per sub-system. Hybrid patterns could have effects on quality attributes, which are not described in this research. 2. The MAAM is not evaluated yet in an extensive industrial setting. By performing an industrial evaluation, the applicability of the model can be validated to a larger extend.

All are threats to *external* validity, as defined by Yin (2009) and are not critical. We suggest further research to focus on demonstrating the analytic hierarchy process in conjunction with the decision matrix at several companies. Then, the ratings can be evaluated more thoroughly resulting in possible adjustments for these performance values. Furthermore, the ratings provided in this research are based on subjective judgements of sixteen experts. The accuracy of the ratings can be increased by surveying a larger number of experts, causing a decrease of the standard deviation. Finally, combinations of the MTA patterns (i.e. hybrid patterns) should be evaluated to check for potential unexpected effects on software quality.

Chapter 7

Comparing Dynamical Adaptation Patterns

Abstract

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

This work has been published as *Comparing Two Architectural Patterns for Dynamically Adapting Functionality in Online Software Products* in the Proceedings of the 5th International Conferences on Pervasive Patterns and Applications (PATTERNS 2013) (Kabbedijk, Salfischberger, and Jansen, 2013) and is extended as a journal submission. It is co-authored by Tomas Salfischberger and Slinger Jansen

7.1 Introduction

Software as a Service (SaaS) is a rapidly growing deployment model with a clear set of advantages to software vendors and their customers. SaaS allows vendors to deploy changes to applications more rapidly, which increases product innovations while reducing support-costs as only a single version is to be supported concurrently (Dubey and Wagle, 2007). In the SaaS deployment model a single application serves a large number of customers. These customers are called tenants, which can be a single user or an organisation with hundreds of users. Because all tenants use the same application, the cost of development and setup of the application can be amortized over all contracts.

The multi-tenant deployment model requires the application to be aware of different tenants and their users, for example in separating the data visible to different groups of users. We define multi-tenancy as: "the property of a system where multiple varying customers and their end-users share the system's services, applications, databases, or hardware resources, with the aim of lowering costs". Database designs for multi-tenant aware software require specialized architecture principles to accommodate multiple tenants (Aulbach, Grust, Jacobs, Kemper, and Rittinger, 2008). One of the challenges in multi-tenant application architectures is the implementation of tenant-specific requirements (S. Jansen, 2010). Variability of software to fit requirements of specific customers can no longer be done by directly modifying the application for each client, because a single application serves multiple customers.

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

The concepts of variability and quality attributes are explained in Section 7.2, after which the expert evaluation used is explained in Section 7.3. The Component Interceptor Pattern and the Event Distribution Pattern, two patterns both solving the problem of dynamically adapting functionality of online business

software, are presented in Section 7.5. Section 7.6 presents the DATASOURCE ROUTER PATTERN and CUSTOM PROPERTY OBJECT PATTERN, which introduce variability in the datamodel of online software products. All patterns are compared in terms of security, performance, scalability, maintainability and implementation effort. A concluding overview, presenting the best suitability for all patterns van be found in Section 7.7.

Please note; in the text, we set pattern names in SMALL CAPS according to the convention by Alexander, Ishikawa, Silverstein, Jacobson, Fiksdahl-King, and Angel (1977).

7.2 Related Work

Software Patterns - Object oriented design patterns were first introduced by Gamma, Helm, Johnson, and Vlissides (1995) who define design patterns as recurring patterns of classes and communicating objects in many object-oriented systems. They state "each design pattern systematically names, explains, and evaluates an important and recurring design in object-oriented systems". We distinguish the patterns described in this research from the original object oriented design patterns by using the name software patterns and define them as "A particular recurring design problem that arises in specific design contexts, and presents a well-proven generic scheme for its solution. The solution scheme is specified by describing its constituent components, their responsibilities and relationships, and the ways in which they collaborate." (Buschmann, Meunier, Rohnert, Sommerlad, and Stal, 1996, p. 8). We intend to describe software patterns for variability techniques in a multi-tenant context in a similar manner to the object oriented design patterns described by Gamma, Helm, Johnson, and Vlissides (1995).

Others have, based on the first set of design patterns by Gamma, Helm, Johnson, and Vlissides (1995), researched the best methods for describing and communicating design patterns for later reuse. For example Evitts and Hinchcliffe (2000) applies the UML to design patterns and proposes a modeling technique based on UML-modeling. The same approach is taken by Mapelsden, Hosking, and 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 linking the elements of design patterns in DPML to UML model elements. Mapelsden, Hosking, and Grundy (2002) 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 (Maplesden, Hosking, and Grundy, 2001).

Lauder and 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, Helm, Johnson, and Vlissides (1995). The proposed solution is to apply three separate layers of modeling, the role-model, type-model and class-model. At the highest level of modeling the role-model only describes the parts of a design pattern and their relative roles and interaction. The type-model is a refinement of the role-model where details like implemented methods are added. The type-model should according to Lauder and Kent (1998) be supplemented by a textual description of the motivation, trade-offs and known uses. The final refinement of the type-model is the class-model where a concrete implementation is described as is the case in previous pattern description languages.

Variability - The field of software variability has been the subject of research from both the modeling perspective as well as the technical perspective. Software variability modeling is common in software product lines as described by Jaring and Bosch (2002). The application of variability modeling as used in product line variability (Bayer, Gerard, Haugen, Mansell, Møller-Pedersen, Oldevik, Tessier, Thibault, and Widen, 2006) to software as a service environments has been described by Mietzner, Unger, Titze, and Leymann (2009). Variability modeling as discussed in the aforementioned works contributes to the understanding of where the application architecture needs to be able to accommodate change or extension. Patterns play an important role in modeling and solving variability in software products (Kabbedijk and Jansen, 2012).

Svahnberg, Gurp, and Bosch (2005) propose feature diagrams as a modeling technique to describe the different variants of feature in a software product. Svahnberg, Gurp, and Bosch (2005) use their feature diagrams as the basis for a method to

identify variability in a product, constrain this variability, pick a method of implementation for the variability and further manage this variability point in the application lifecycle. The main difference from the objectives of our research is that Svahnberg, Gurp, and Bosch (2005) describe implementation techniques for variability per installation instance of the software, whereas we focus on *runtime* variability in a multi-tenant context.

Quality Attributes - Benlian and Hess (2011) identify *security* as one of the most important risk-factors perceived, followed by performance risks. To assess security risks, SaaS vendors need to include security as a quality attribute in their design of the architecture. This leads to security as the first desired quality attribute for business SaaS. *Performance* as an important factor to SaaS users is closely related to the most important factor as found; cost Benlian and Hess (2011). When performance is insufficient, clients are lost, when the system uses too many resources to gain an acceptable level of performance, cost is increased. A SaaS vendor must thus assess the possible performance impact of changes to the software. To control cost in business SaaS, the SaaS vendor needs to utilize its opportunities for scalability to decrease the cost of hardware or hosting fees (e.g. using scalable software to make optimal use of cloud-hosting).

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

The identified quality attributes are the following:
Security - The ability to isolate tenants from each other and the possible impact of security breaches in custom components on other parts of the system.
Performance - The utilization of computing, storage and network resources by the application at a certain level of usage by clients.
Scalability - The relative increase in capacity achieved by the addition of computing, storage and network resources to the system as well as the flexibility with which these resources could be added to the system.
Maintainability - The ease with which the system can be extended and potential

problems can be solved.
Implementation Effort - The effort required to implement and deploy a specific system.

7.3 Research Approach

In order to identify the patterns in this chapter, a design science approach (Hevner and Chatterjee, 2010) was used in which patterns are constucted based on variability solutions observed in different software products. The candidate patterns are evaluated by experts to improve the pattern description and ensure the correctness. A holistic multiple case study design was applied in which three case companies were used to gather the patterns. Company A has a software product for high volume analysis of marketing data, Company B has a customer relations management product for small to medium sized companies and Company C has a logistics planning product for complex supply chains. The observed variability solutions are all implemented in current commercial software products and one of the researchers took part as a consultant in all three case companies. Solutions observed in at least two independent products are considered to be general solutions and presented as patterns. All patterns are evaluated by two experts to enhance the validity of the patterns (Runeson and Höst, 2009). During each evaluation session, a pattern is discussed with an expert, in a semi-structured way. The effect of the patterns in relation to a set of quality attributes is discussed per attribute, after which additional topics related to the patterns that came up during the interview (e.g. implementation considerations), are discussed. The results are used to validate the patterns and to evaluate the consequences of implementing the patterns.

Both experts used in this study have experience in the development of large enterprise software products and multi-tenancy. The first expert used within this research to evaluate and validate the results, is a senior software architect in an international software consulting firm specialized in large scale development of Enterprise Java applications. His role is to investigate technologies and methodologies to help design better architectures resulting in faster development and more extensible software. A recent project includes a multi-tenant administrative application storing security sensitive data for multiple organizations. The second expert is a technology director and lead architect for an application used in

distributed processing of data, previously working in software performance consulting for web-scale systems. His experience lies in the field of high-performance distributed computing. The application his company works on focuses of low-latency coordinated processing of large volumes of data to calculate metrics used for marketing.

7.3.1 Validation

The patterns identified within this chapter are evaluated by two experts, who validated all patterns based on correctness, completeness and understandability of description. By using the two experts to validate the patterns, the likeliness of capturing the appropriate problems and solutions in the patterns (i.e. construct validity), is enhanced. Additionally, the pattern descriptions are reviewed by the case company architects, in order to confirm the descriptions accurately describe the actual implemented solution. The review of the patterns has a positive effect on the external validity of the results. Finally, the fact that all patterns are observed in multiple case companies, also enhances the construct validity by using multiple sources of evidence (Yin, 2009).

7.4 Pattern Description Method

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

The needs for a description language for the discussed design patterns thus includes the need to describe any necessary characteristics of the supporting systems and auxiliary materials. When considering design patterns for software systems we propose a combination of description techniques at different levels similar tot Lauder and Kent (1998). Instead of modelling different levels of detail and abstraction

within only object oriented design different levels of the software architecture including supporting systems have to be modelled. The levels we propose to describe online systems are:

1. Functional level
2. System level
3. Implementation level

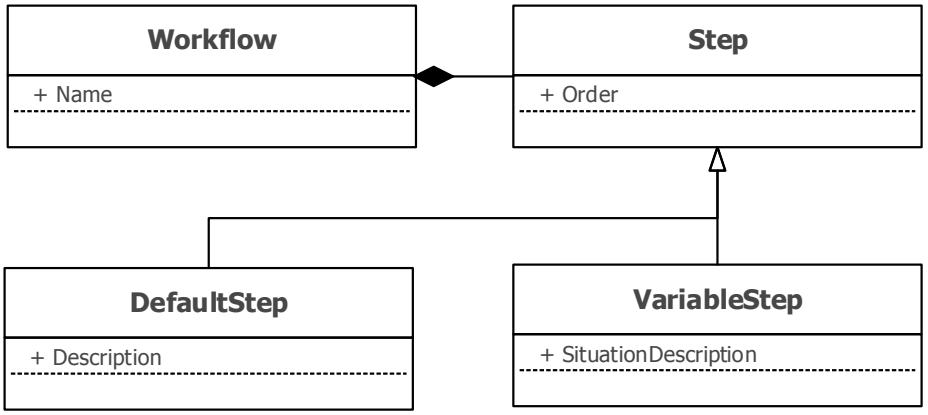


FIGURE 7.1: Example UML class diagram

Functional level - This level describes the functional intention of the pattern in a technical context. Multiple different patterns can share the same model at functional level, because several patterns can be designed to reach the same functional effect with, for example, different performance and scalability characteristics. For the graphical modelling of the functional level, UML class diagrams are used as shown in Figure 7.1. This diagram captures the functional situation resulting from application of the pattern without considering implementation of pattern instantiation details.

System level - This level models the overview of the software including supporting systems after the application of the pattern. Interaction among different components within and between systems as a result of the implemented pattern are shown. A UML deployment diagrams (Rumbaugh, Jacobson, and Booch, 2004) is used to describe this level (see Figure 7.2 for an example).

Implementation level - The third level describes the potential implementation of the pattern. These diagram depicts a specific implementation of the components of the pattern. The implementation diagram is closely related to the system model,

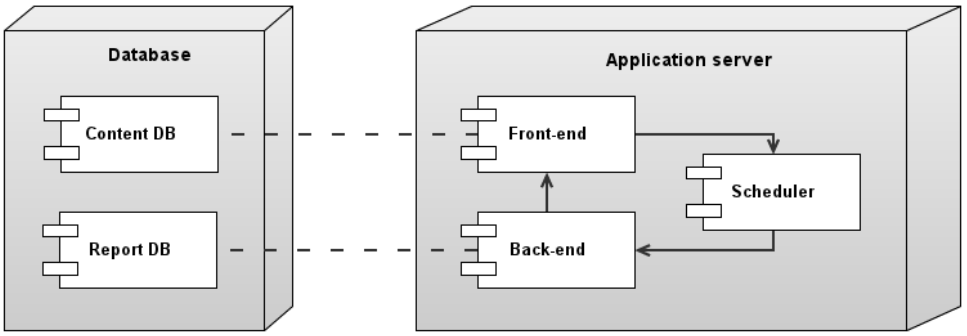


FIGURE 7.2: Example UML Deployment Diagram

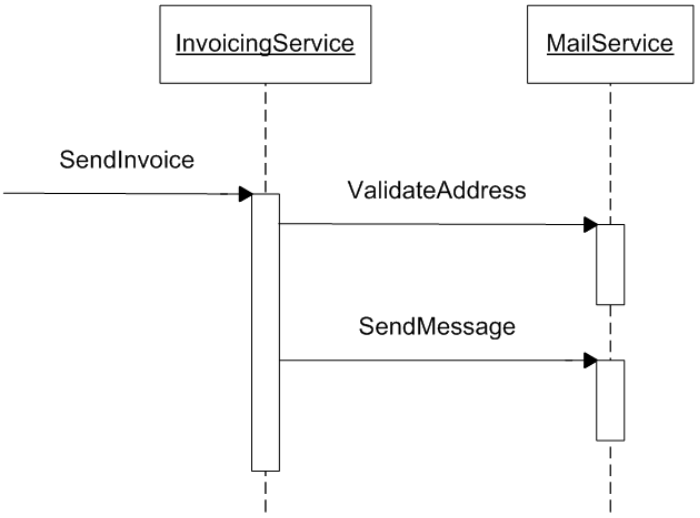


FIGURE 7.3: Example Sequence Diagram

but depicts the method of application of the components in the system model on a more detailed level. Within this research we use a sequence diagram as shown in Figure 7.3 to illustrate the implementation. This description level should be regarded as a possible way to implement the pattern, but it does not prescribe a specific implementation.

This chapter applies an alternative pattern description model than presented in Chapter 3. The reason is that in order to assist architects in implementing the patterns a lower abstraction level is required in the pattern descriptions. The pattern catalogue in Appendix A, however, does present the patterns from this chapter formatted according to the description model from Chapter 3.

7.5 Dynamic Functionality Adaptation Patterns

7.5.1 Problem Statement

Software product vendors not only need to offer a *data model* that fits an organisation's requirements, *software functionality* also has to meet an organisation's processes (Van der Aalst, Hofstede, and Weske, 2003). When tailor-made software is developed, it is possible to set the requirements to exactly match the processes of a specific organisation. For standard online software products this is not possible and differences between requirements of organisation have to be addressed at runtime.

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

To allow for the addition of extra functionality in the application a solution is needed that allow to configure this functionality. This functional situation is modeled in Figure 7.4, the envisioned functional situation. The *StandardComponent* is a normal component of the software with default functionality, this component has a set of *ExtensionPoints*. An *ExtensionPoint* is a location within the normal workflow where there is a possibility to add or change functionality. This functionality is specified in an *ExtensionComponent*, which contains the actual functionality that is to be executed at the specified *ExtensionPoint*.

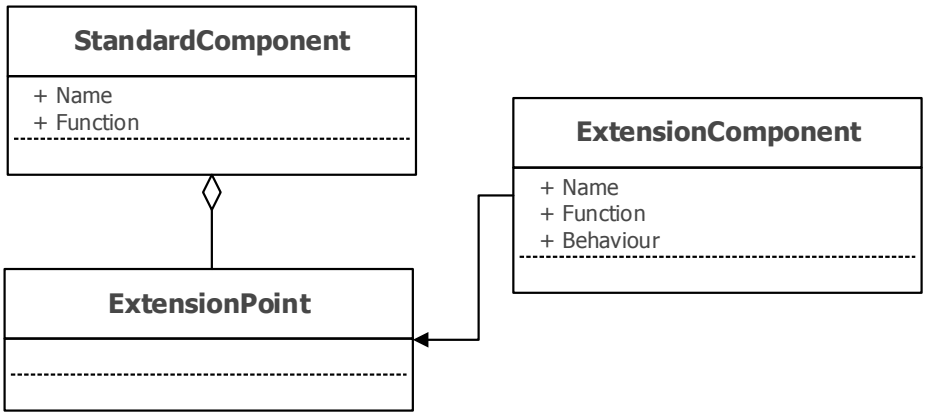


FIGURE 7.4: Functional Model for adapting functionality

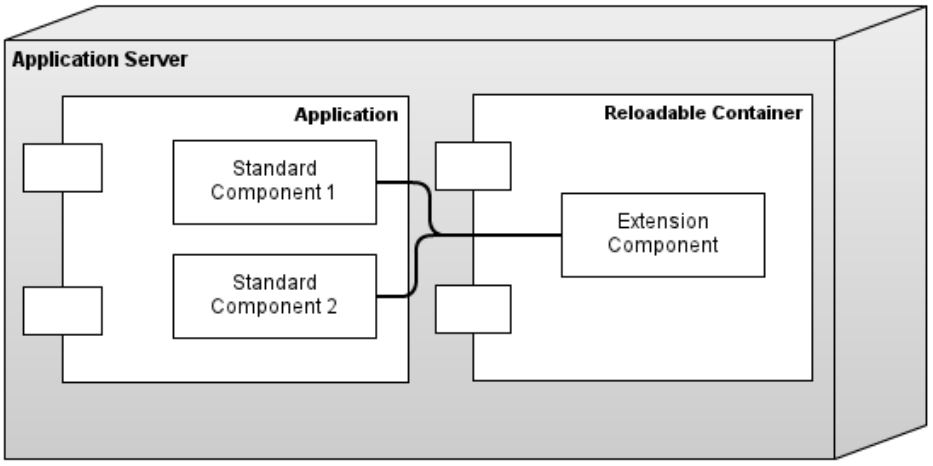


FIGURE 7.5: Component Interceptor Pattern: System Model

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

7.5.2 Component Interceptor Pattern

The COMPONENT INTERCEPTOR PATTERN as depicted in Figure 7.5 consists of only a single application server. Interceptors are tightly integrated with the application, because they run in-line with normal application code. Before the *StandardComponent* is called the interceptors are allowed to inspect and possibly modify the set of arguments and data passed to the standard component. To do this the interceptor has to be able to access all arguments, modify them or pass them along in the original form. Running interceptors outside of the application requires marshalling of the arguments and data to a format suitable for transport, then unmarshalling by the interceptor component and again marshalling the possibly modified arguments to be passed on to the standard component that was being intercepted. This is impractical and involves a performance penalty (Carpenter, Fox, Ko, and 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 form of a reloadable container. In some implementations this could be as simple as changing a source file, because the programming platform used will interpret source code on the fly. Other platforms require special provisions for reloading code, such as OSGi for the Java platform or Managed Extensibility Framework for the .NET platform.

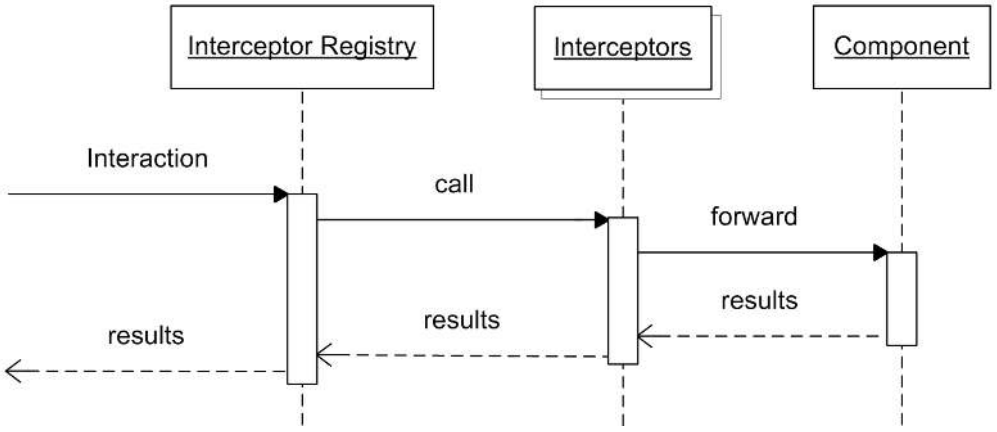


FIGURE 7.6: Component Interceptor Pattern: Sequence Diagram

Figure 7.6 depicts the interaction with interceptors involved. Interaction with standard components that can be extended goes through the interceptor registry. This registry is needed to keep track of all interceptors that are interested in each interaction. Without the registry the calling code would have to be aware of all possible interceptors. As depicted, multiple interceptors can be active per component. It is up to the interceptor registry to determine the order in which interceptors will be called. An example strategy would be to call the first registered interceptor first or to register an explicit order when registering the interceptors.

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

7.5.3 Event Distribution Pattern

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

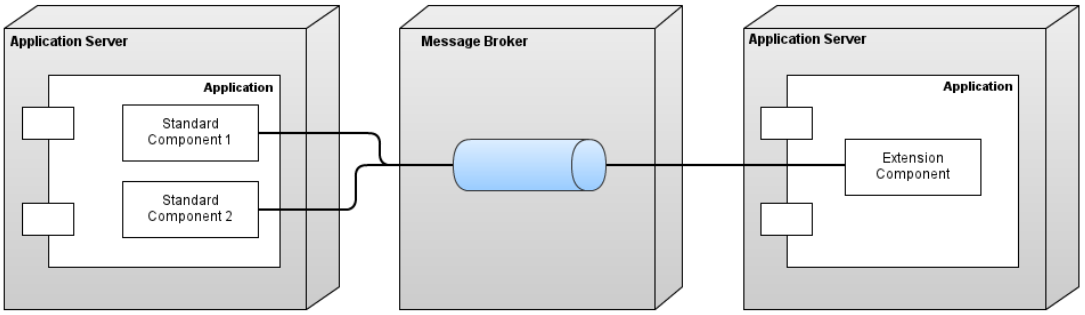


FIGURE 7.7: Event Distribution Pattern: System Model

In the example of a *ClientUpdated* event an extension component could be developed that sends a notification to an external system to update the client details there.

The system model in Figure 7.7 depicts the distributed nature of the EVENT DISTRIBUTION PATTERN. Standard components run in the application server, sending events to a central broker, which can be run outside of the application. Extension components are isolated and can be on a separate physical server or run as separate processes on the same server depending on capacity and scale of the application. Components are loosely coupled, sharing only the predefined set of events.

The standard components are unaware of which extension components listen for their events, execution of extension components is decoupled from the standard components. Executing the extension components separately allows for independent scalability of these components. Depending on system load and the volume of events each component listens for, it is possible to allocate the appropriate amount of resources to each component. Because there is no interaction between listeners, it is possible to execute all listeners in parallel if appropriate for the execution environment.

Standard components publish events to the broker as depicted in the sequence diagram in Figure 7.8. The activation of the standard component not necessarily overlaps with its listeners. After publishing the event, a standard component is free to continue execution. Depending on the fault tolerance and nature of the events it is up to the standard component to make a trade-off between guaranteed delivery at a higher latency by waiting on the broker system to acknowledge reception of the event or continue without waiting for such an acknowledgement. If, for example, an event is only meant to prime a cache for extra performance the loss of such a message would not impact critical functionality of the system while waiting

for the message might mitigate any performance gains. If on the other hand an event is used for updating an external system for which no other synchronization method is available the system needs guaranteed delivery to function correctly. At design time this decision can be made on an event by event basis depending on the capabilities of the messaging system used.

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

7.5.4 Pattern Comparison

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

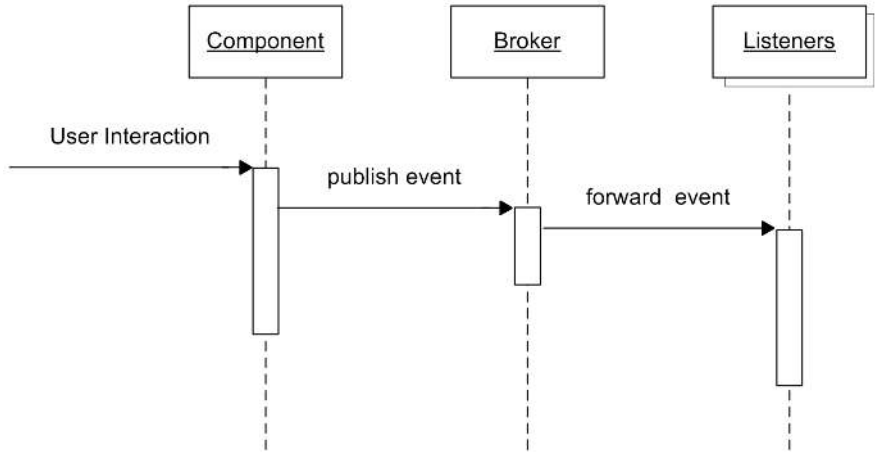


FIGURE 7.8: Event Distribution Pattern: Sequence Diagram

7.5.4.1 Security

When adapting functionality of an application, there is always the possibility of introducing new security vulnerabilities. This is an inherent risk of extending an application. The variability patterns do, however, influence how much larger the attack surface becomes and how well a breach in one of the components is isolated from other components. In the COMPONENT INTERCEPTOR PATTERN the code handling the new functionality becomes part of the application and will have the ability to execute arbitrary code within the context of the main application as depicted in Figure 7.5. It will also have full access to any parameters passed to intercepted functions as well as any returned values. A security breach in the extension components (interceptors) is not isolated to only those components unless extra security measures are implemented to separate the components from the main application. This isolation would however have an impact on performance because of the nature of the integration.

The EVENT DISTRIBUTION PATTERN isolates the extension components from the application by executing them in a separate context based on incoming events as depicted in Figure 7.6. This execution in a separate context allows for more isolation between extension components and the main application components. The components also have far more limited access to standard functionality, because any change the component wants to make has to go through explicitly exported APIs or messages. Combined with event-sourcing, any change to data as a result of custom functionality is fully traceable including the original values (Fowler, 2003).

7.5.4.2 Performance

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

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

7.5.4.3 Scalability

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

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

7.5.4.4 Maintainability

When adapting the functionality of an application, maintainability is also affected by the necessity to make sure future extensions and modifications are compatible with any custom functionality implemented for tenants. This is a trade-off between the flexibility and depth with which *ExtensionComponents* can affect the application and the impact that changes to the application will have on the *ExtensionComponents*. As an example of the aforementioned trade-off a simple system

with only a single *ExtensionPoint* will have a much lower impact on maintainability than a complex system with a very high number of *ExtensionPoints*. This however affects both patterns equally.

The way the patterns decouple *ExtensionComponents* from *StandardComponents* is however a differentiating factor. In the COMPONENT INTERCEPTOR PATTERN the *ExtensionComponent* is more tightly integrated with the *StandardComponent* because calls to a *StandardComponent* at an *ExtensionPoint* go through the interceptor providing all parameters and return values of the call. When changing calls by adding or removing parameters this will directly affect the input of each *ExtensionComponent* registered from that *ExtensionPoint*. When applying the event distribution pattern the integration is more decoupled because calls to StandardComponents are not directly affected by the *ExtensionComponents*. Instead the *ExtensionComponent* receives a standardized event-message and uses a provided API to send any changes or other actions back to the application. This allows for changes to the *StandardComponent* without changing the event-messages going to the *ExtensionComponent*. At the same time the API used by *ExtensionComponents* to influence the application can be kept stable for small changes or versioned to support future compatibility using methods like the one described by Weinreich, Ziebermayr, and Draheim (Weinreich, Ziebermayr, and Draheim, 2007).

7.5.4.5 Implementation Effort

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

In the EVENT DISTRIBUTION PATTERN, a message broker system must be setup to handle the event-messages flowing from *StandardComponents* to *ExtensionComponents*. The application still has to be modified at the ExtensionPoints to send

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

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

Please see Table 7.1 for an overview of the evaluation of both patterns. Plus and minus signs are used to indicate whether a characteristic is positive or negative. Keep in mind all scores are relative scores compared to the other pattern.

7.6 Dynamic Data Model Extension Patterns

7.6.1 Problem Statement

Organisations within the same or different market all strive to differentiate themselves, which results in numerous different working processes each with specific requirements for the supporting software systems. Additionally, across markets and jurisdictions differences exist in regulations and standards which require the storage and reporting of different data for each organisation. Organisations will thus set varying requirements to store data specific to their needs. These requirements could be met by software specifically designed for the market in which this organisation operates or even software tailored to the needs of one specific organisation. Specializing software of a small market or even single organisation decreases the number of possible clients for the software vendor and increases the cost per

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

TABLE 7.1: Overview of both Dynamic Functionality Adaptation Patterns

client. A software product that provides enough variability on the data model to meet organisation specific requirements will decrease cost and attract clients that cannot currently be serviced by software products unable to meet their specific requirements. Extension of the data model by creating additional fields to store data that are specific to an organisation or their working processes is a common requirement (Sun, Zhang, Guo, Sun, and Su, 2008).

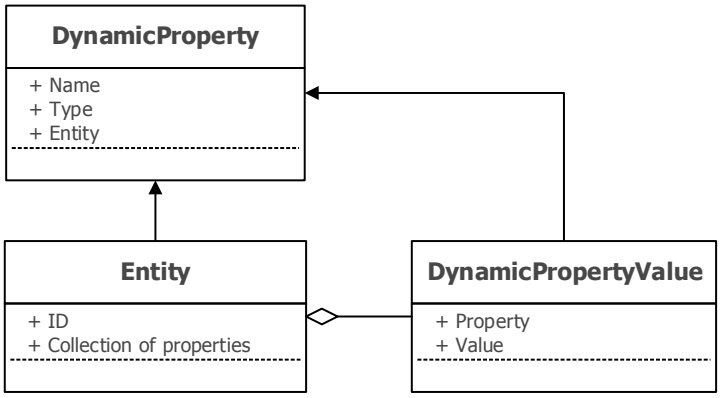


FIGURE 7.9: Functional Model for datamodel extension

In case of standardized software where this requirement is not met by the default installation of the software an extension of the existing data model is required. Figure 7.9 depicts the envisioned functional situation, storing custom properties of entities in the domain model. The depicted *Entity* is the original entity in the application domain model which contains a *DynamicPropertyValue* and has a relation to a *DynamicProperty*. This property is configured for a specific tenant and holds settings like for example a name and expected data-type.

7.6.2 Datasource Router Pattern

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

The system model, which is shown in Figure 7.10, describes the overview of the system when implementing the DATASOURCE ROUTER PATTERN. As shown, the

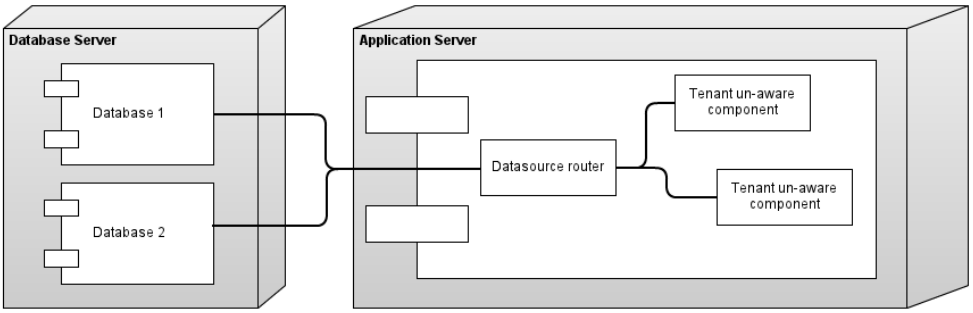


FIGURE 7.10: Datasource Router Pattern: System Model

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

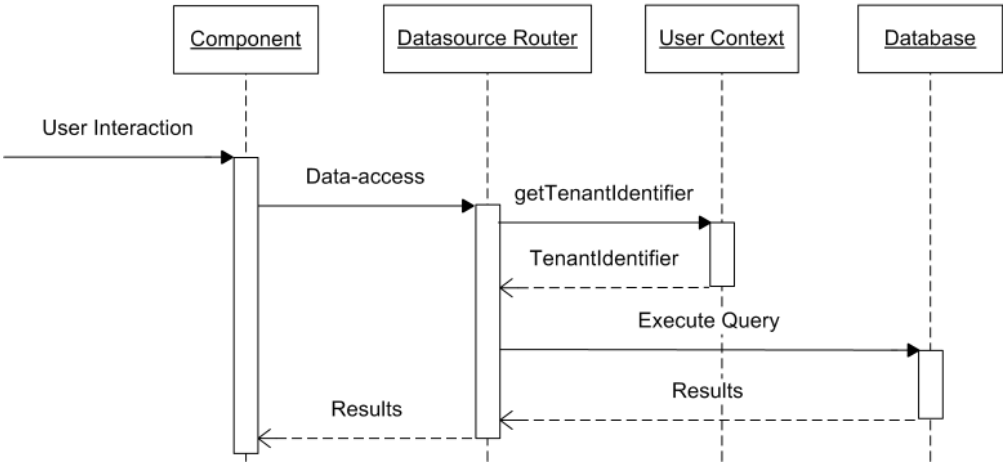


FIGURE 7.11: Datasource Router Pattern: Sequence Diagram

The interaction between tenant-unaware components and the database goes through the *Datasource Router*. The sequence diagram in Figure 7.11 depicts the interaction from component through *Datasource Router* to the actual database. First the user interacts with a component, this component requires access to data which is done through the *Datasource Router*. The *Datasource Router* is then responsible for determining which tenant the current user belongs to, this responsibility is delegated to the *User Context*. It is implementation dependant how this *User*

Context is implemented, the only requirement is that it is able to tell the *Datasource Router* which tenant is to be used in the context of the current request. After determining which tenant is active the *Datasource Router* executes the query on the right database (selected based on the active tenant), the results are then returned to the component which originally needed access to the data. In this sequence it is clear that from the perspective of a component requesting data it does not matter how multi-tenancy is implemented in deeper layers. The component is isolated from these choices and the possible complexity involved in selecting the right datasource to use for the current user.

7.6.3 Custom Property Object Pattern

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

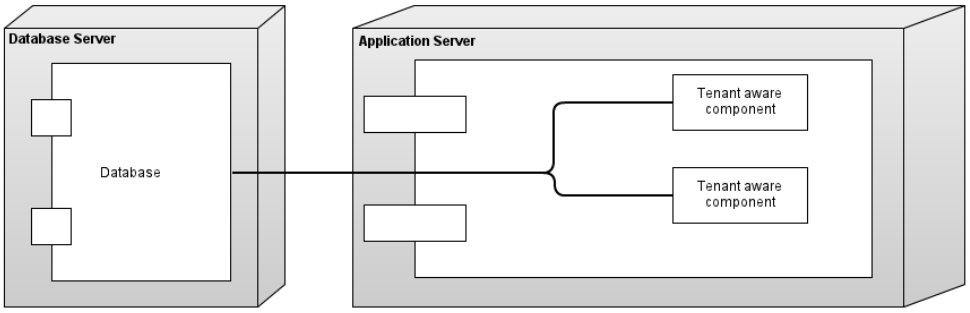


FIGURE 7.12: Custom Property Object Pattern: System Model

This pattern prescribes the storage of all data in a single database which is accessed by components that are aware of how to filter data for each tenant. In the system model, as depicted in Figure 7.13, components are aware of multi-tenancy and directly access a single database to query for the data necessary to complete requests. When querying the data it is the responsibility of each component to only query data related to the requested tenant or filter data while processing, to get results only for the current tenant.

As a result of using a single database for all tenants, the other components need to be aware of the context in which they operate. When retrieving data the

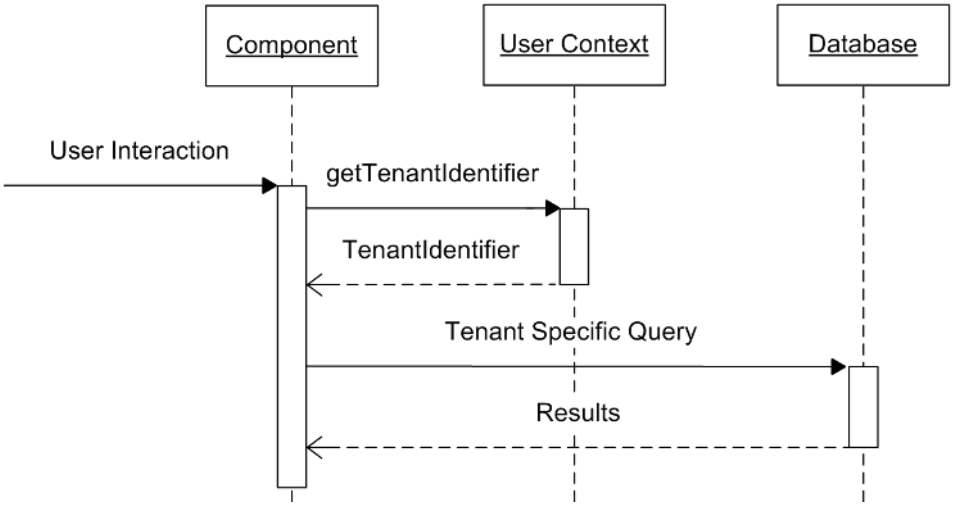


FIGURE 7.13: Custom Property Object Pattern:Sequence Diagram

components need to filter the results to only show data for the current tenant. The resulting interaction from component to database is depicted in Figure 7.13. The component first determines which tenant is currently active, this is done by using the User Context. It is implementation dependant how this User Context determines this, the only requirement is that it is able to tell a component which tenant is to be used in the context of the current request. The component then generates a query that is specific to the current tenant and sends this to the database. It is the responsibility of the component to ensure that the generated query only accesses data for the current tenant and to avoid retrieving data outside of tenant boundaries.

7.6.4 Pattern Comparison

7.6.4.1 Security

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

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

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

7.6.4.2 Performance

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

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

for the DATASOURCE ROUTER PATTERN, because the data for each tenant is stored separately.

7.6.4.3 Scalability

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

By applying the CUSTOM PROPERTY OBJECTS PATTERN the application will only use a single database system. This impacts scalability in the application 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 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 be stored on the new systems. There is virtually no overhead involved in this addition, because no extra coordination is required between the database systems servicing data for separate tenants.

The CUSTOM PROPERTY OBJECTS PATTERN requires a database system that is able to store all data for all tenants. The database system must in that case support vertical scalability by increasing the capacity of a single system instead of horizontal scalability. The application of a database system that provides a

scalability capability is necessary for large deployments of this pattern. The results are dependant upon the effectiveness with which the database system deals with scalability challenges.

7.6.4.4 Maintainability

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

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

7.6.4.5 Implementation Effort

For the DATASOURCE ROUTER PATTERN the initial implementation requires the development of the router component as well as systems to manage and automatically deploy new database instances for new tenants. The other components can however be left unchanged because awareness of the multi-tenant environment is not required. Using the CUSTOM PROPERTY OBJECTS PATTERN, on the other

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

Table 7.2: Overview of both Dynamic Datamodel Extension Patterns

hand, does not require the development of new components or management systems. For this pattern the existing components need to be adapted to query the right data and use appropriate filtering methods. Both patterns require the implementation of code handling the existence of custom properties for entities in the applications data model. This is equal for both patterns and thus of no influence in a comparison on implementation effort.

7.7 Conclusion

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

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

Second, two dynamic data model extension patterns, being the DATASOURCE ROUTER PATTERN and CUSTOM PROPERTY OBJECT PATTERN are presented and evaluated. We conclude that the DATASOURCE ROUTER PATTERN has advantages on *security* by naturally isolating the data for all tenants, *scalability* by allowing for

the distribution of tenants across datasources and *implementation* by not requiring all queries and components to be adapted but providing a single router component instead. The custom property objects pattern holds an advantage on performance by allowing better resource utilization, however extra care is necessary to design an appropriate database schema. The CUSTOM PROPERTY OBJECTS PATTERN also scores better on *maintainability* by allowing standardized handling of the dynamic properties and using a static data model avoiding the need to test every possible variation when adapting the software.

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

Chapter 8

Software Pattern Evaluation Method

Abstract

Software architecture makes extensive use of many software patterns. The decision on which pattern to select is complex and architects struggle to make well-advised choices. Decisions are often solely made on the experience of one architect, lacking quantitative results to support the decision outcome. There is a need for a more structured evaluation of patterns, supporting adequate decision making. This paper proposes a Software Pattern Evaluation Method (SPEM) that enables the quantification of different pattern attributes by using structured focus groups. The method is formed using a design science approach in which an initial method was created using expert interviews, which was later refined using several evaluation sessions. SPEM helps software producing companies in structuring their decision making and selecting the most appropriate patterns. Also, SPEM helps in enriching pattern documentation by providing a way to add quantitative information to pattern descriptions.

This work has been published as *SPEM: A Software Pattern Evaluation Method* at the 6th International Conferences on Pervasive Patterns and Applications (PATTERNS 2014) (Kabbedijk, Donselaar, and Jansen, 2014). It is co-authored by Rene van Donselaar and Slinger Jansen

8.1 Introduction

Modern software architecture heavily relies on the use of many different software patterns, often used complementary to each other in order to solve complex architectural problems. Software architecture provides guidelines and tools for high-level system design in which architects select best fitting patterns to be used within the software product (Bass, Clements, and Kazman, 2013). Many different patterns and tactics exist, leading to a complicated trade-off analysis between different solutions and causing the evaluation and selection of the appropriate software patterns to be a complex task (Jansen, Van Der Ven, Avgeriou, and Hammer, 2007). This complexity means architects need to have in-depth understanding of the project characteristics and requirements combined with extensive experience in software development.

The information needed for appropriate pattern selection is seldom available to all architects in a centralized or standardized way. Architectural decisions are frequently still made based on experience and personal assessment of one person, instead of using the knowledge of many (Babar and Gorton, 2007). Allowing software architects to use all information efficiently saves time when selecting fitting software patterns and leads to better and more adequate decision making. For this to be possible, a method has to be created, enabling the evaluation and documentation of crucial attributes of a software pattern (Tyree and Akerman, 2005). This structured evaluation will allow architects and decision makers to compare different solutions and select the best matching pattern. Patterns, however, are an high-level solution that can be used different scenarios, making it impossible to use one specific implementation of the pattern to evaluate the entire pattern. Because specific implementations are unusable, the relevant pattern attributes can not be directly measured in a quantitative way.

Pattern evaluation adds retrospect and the knowledge of many experts to existing pattern documentation. This study also relates to software architecture as it solves a problem found in the software pattern selection process. Software pattern evaluation helps when performing pattern-oriented software architecture in cases where alternative patterns to solve the same problem and only a single pattern can be selected. This is an important factor to take into account, because it means that rather than selecting individual patterns, an architect will want to select an architectural style, and thus select a large set of patterns that fit this style. This

area of software architecture has developed, which resulted in a large amount of documented patterns and allows for comparing architectural styles (Booch, 2005).

Although it seems that comparing individual patterns is less relevant for software architecture, an architectural style is selected at the early stages of software design and cannot easily be changed after the development has started. This creates a problem because while the software is being developed, the requirements for the project or the environment will change. Therefore, it is necessary to extend the architecture or at times alter the existing architecture. At this point, it becomes relevant to compare individual patterns in order to select the pattern that fits the project requirements. This is an ongoing process that happens throughout software development and relies on the experience of software architects and developers. Current documentation of software patterns lacks a way to compare them with each other. But if multiple patterns tackle the same problem, how does an architect decide which one to use? This is tacit knowledge of experienced architects and developers, leading to the following problem statement:*"There is no formal way to express the quality of one pattern over another"*.

This paper presents the Software Pattern Evaluation Method (SPEM). Using SPEM, software producing companies are supported in pattern selection decision making and are able to quantitatively compare different patterns. SPEM enables them to get an overview of specific pattern characteristics in a timely manner. Also, SPEM can be used to generate a publicly available pattern related body of knowledge, helping research and practitioners in architectural research and decision making. This paper first gives an overview of research related to pattern comparison in Section 8.2. The design science approach used in the research is described in Section 8.3, after which SPEM is presented in Section 8.4. The pattern evolution, including the initial method creation (Section 8.5.1) and method evaluation (Section 8.5.2), showing the changes during the method creation process can be found in Section 8.5. To conclude, the application of SPEM are discussed (Section 8.6), followed by a conclusion (Section 8.7).

8.2 Related Work

Software Patterns — As software development was maturing in the 1980s, the need arose to share common solutions to recurring problems. This process started out by developers communicating to their colleagues how they solved a recurring development issue. The communication was informal, and there were no clear

rules for documentation. In later years software patterns have become an essential part of software development as a way to capture and communicate knowledge. Software patterns are solutions to a recurring problem in a particular context (Buschmann, Meunier, Rohnert, Sommerlad, and Stal, 1996; Gamma, Helm, Johnson, and Vlissides, 1995). When properly documented, these solutions are a valuable asset for communication with and among practitioners (Beck, Crocker, Meszaros, Vlissides, Coplien, Dominick, and Paulisch, 1996). Usage of software patterns allows for time and cost reduction in software development projects, making them an important tool for software design and development. Although software patterns started out as a way to communicate solutions among developers, they have become a crucial part of software architecture (Buschmann, Henney, and Schmidt, 2007c) as well. A pattern selected by a developer, however, does not take into account the entire architecture and how it combines with existing patterns. This problem is solved by selecting patterns at the architectural level.

Architecture Evaluation — Evaluation is commonly used in software architecture in order to increase quality and decrease cost (Abowd, Bass, Clements, Kazman, and Northrop, 1997). Many evaluation methods for software architecture have been developed and compared in recent years (Babar, Zhu, and Jeffery, 2004). The evaluation should be performed as early as possible in order to prevent large-scale changes in later stages of development. Software architecture evaluation is linked to the development requirements and desired quality attributes. Therefore, it is not a general evaluation of software architecture, nor an evaluation of a specific implementation. The evaluation should be an indication of whether the proposed architecture is a good fit for the project. Pattern comparison and evaluation has been done before in a quantitative manner (Hills, Klint, Van Der Storm, and Vinju, 2011), but has focussed on the *implementation* of different patterns and lacks the evaluation of the *idea* the pattern describes.

8.3 Research Approach

This section presents the research questions answered in this paper and the design science approach used to construct SPEM. The main research question (MRQ) answered in this paper is:

> ***MRQ:*** *How can software patterns be transparently evaluated and compared during the enterprise software architecting process?*

The aim of this study is to aid software architects in the decision-making process of selecting software patterns. This can only be useful when the pattern evaluation method yields transparent and comparable results. Since software patterns can not be measured objectively, the opinions of multiple software architects are used to form a quantitative and weighted score, representing their common view on the specific pattern. A quantitative study also allows for easy comparison between alternative patterns. For the purpose of answering this research question, multiple sub-questions are constructed:

> ***SQ1:*** *Which quality attributes and characteristics are relevant in pattern evaluation?*

Rationale: Patterns can possess many attributes that give important information on usefulness and quality. For example, how the pattern effects performance or maintainability can both be attributes of a pattern.

Attributes are used in software architecture to evaluate the quality of certain aspects of the architecture. We apply the same principles for evaluation of software patterns. The first step is to create a list of attributes by looking at related literature. This list is then reduced by performing expert interviews. This tells us which of the listed attributes are important to software architects when evaluating a pattern. A validation of the reduced list of attributes is performed by interviewing a second expert. If validation fails, another expert interview and validation is performed. When successful, the result of these interviews is a validated list of attributes that play a role in the software pattern evaluation process.

> ***SQ2:*** *How can attributes relevant in pattern evaluation be quantified in a manner that allows for comparison?*

Rationale: Typical documentation on software patterns is qualitative in nature. Although this might be suited for documentation on patterns it does not allow for comparison. For this reason, the different attributes relevant for pattern evaluation need to be quantified. A structured method of quantification that is used for evaluation would allow for patterns to be compared on attribute level.

To answer the main research question, first comparable methods in which attributes are quantified, within the domain of software engineering, are assessed. From these methods, the specific characteristics are deduced. An example of these characteristics can be the ability to assign a negative value to an attribute. Finding

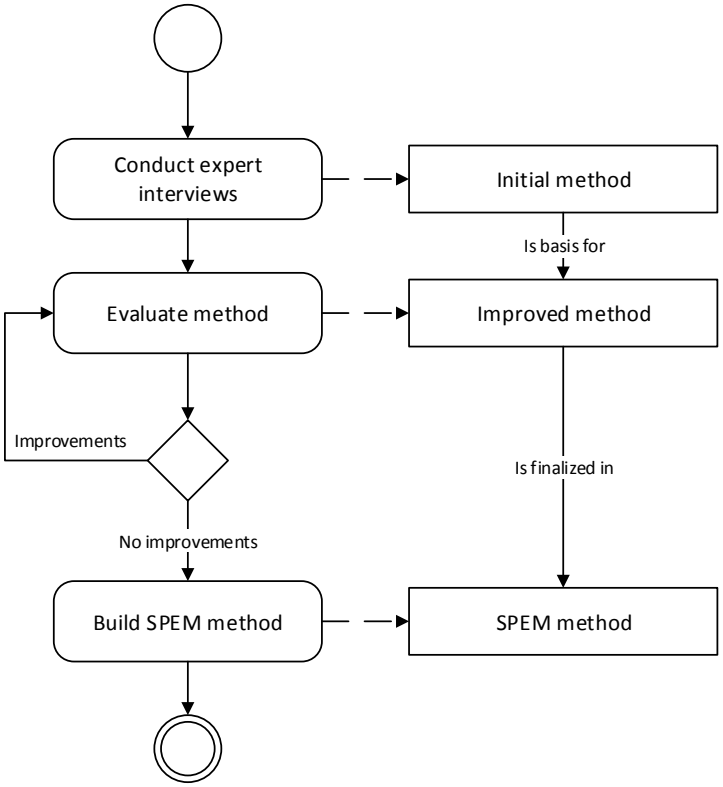


FIGURE 8.1: Design Science Research Method

out which characteristics are important to architects when evaluating a pattern is the next step. This is done by conducting an expert interview. In this interview, the software architect can express which characteristics are important and why. A second interview is held with a different expert to validate the findings. The result is a validated list of characteristics that are important for quantification of attributes. A method for quantification is constructed based on the list of characteristics. The method is evaluated by using it in a focus group session after which is can be incrementally improved.

A design science approach is used, which is depicted in Figure 8.1. An initial method is created based on an earlier exploratory study on the use of focus groups in pattern evaluation (Kabbedijk, Galster, and Jansen, 2012), extended by expert interviews. The method is evaluated in multiple cycles in which the method was put to practice in a real-life setting. Three subsequent sessions are organized in which both professional software architects with a high level of experience and participants with a low level of experience used the method. Software architecture students are used to test how the method functions for participants with a low level of experience. Since SPEM has the aim of aiding software architects with different backgrounds and levels of experiences, it is important to validate the

method with both experienced and inexperienced participants. Also, the sessions with the inexperienced participants are used as pilot sessions, to test the feasibility of the method in practice. The audio and video of all sessions is recorded to be able to analyze the sessions afterwards. Additionally, an evaluation form is filled in by all participants after each session. A revised method was constructed after each session, based on the feedback, which is used in the next session. After three sessions no significant changes were needed based on the feedback, leading to the creation of the final method (i.e. SPEM).

8.4 SPEM - Software Pattern Evaluation Method

SPEM has been constructed to evaluate software patterns in a manner that allows for comparison. There are two distinct roles:

Evaluator — Leads the evaluation process by introducing concepts and directing discussions. He is responsible for timekeeping, collecting all deliverables and noting scores.

Participant — A software architect or developer who uses his knowledge to assign scores to attributes, enters the discussion, shares arguments and tries to reach consensus.

The evaluation data is gathered during a focus group session. These sessions vary in duration from one to two hours. Four to twelve participants can partake in the evaluation, excluding the evaluator. The basis of the evaluation are attributes, categorized in both quality attributes and pattern attributes. Quality attributes are used to measure the impact the pattern has on software quality and are based on ISO/IEC 25010 (ISO/IEC, 2011). The following quality attributes are used in SPEM:

- **Performance efficiency** — Degree to which the software product provides appropriate performance, relative to the amount of resources used, under stated conditions.
- **Compatibility** — The ability of multiple software components to exchange information or to perform their required functions while sharing the same environment.
- **Usability** — Degree to which the software product can be understood, learned, used and attractive to the user, when used under specified conditions.

- **Reliability** — Degree to which the software product can maintain a specified level of performance when used under specified conditions.
- **Security** — The protection of system items from accidental or malicious access, use, modification, destruction, or disclosure.
- **Maintainability** — Degree to which the software product can be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications.
- **Portability** — Degree to which the software product can be transferred from one environment to another

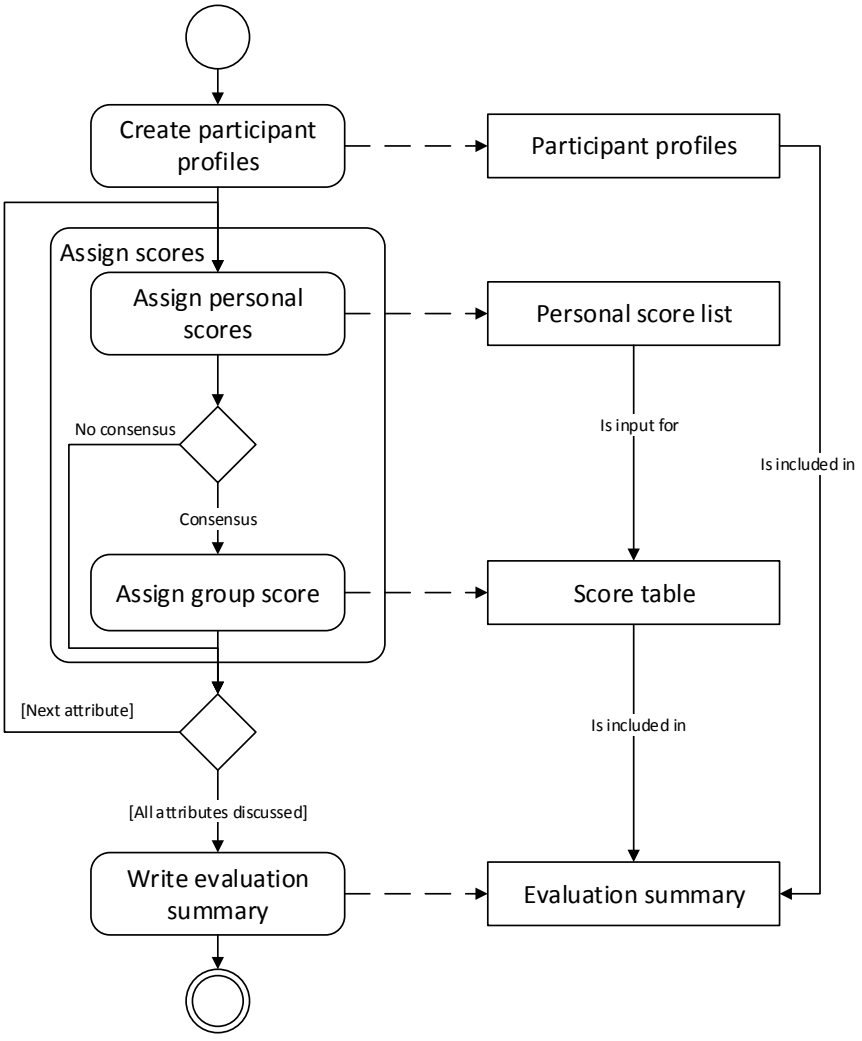
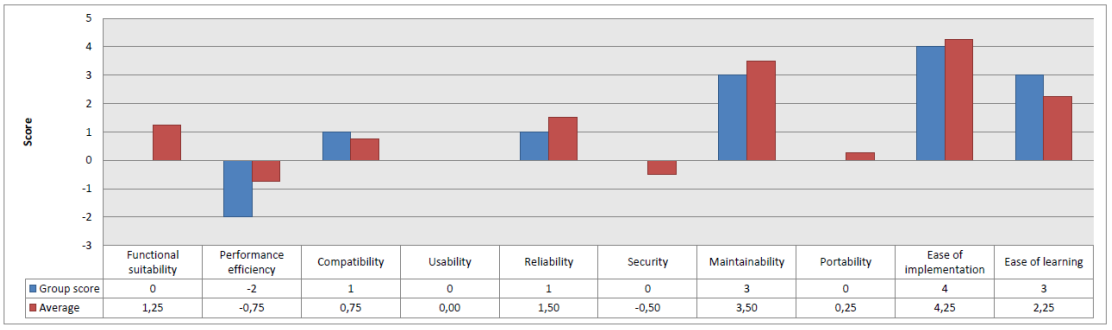


FIGURE 8.2: SPEM: Software Pattern Evaluation Method

Pattern attributes are characteristics of the pattern itself, used to measure its learnability or ease of implementation. The goal of the evaluation is to assign a score to each attribute by all participants. The score is a relative measure based

on the experience of the participant, ranging from −3 to +3. The score is a generalization of the software pattern, not based on a specific implementation. Experience using the pattern in a variety of situations is expressed by the score. Therefore, the difference in experience among all participants is a key factor in the evaluation, which is compensated in a group score. A group score is assigned to each attribute (excluding sub-attribute) and expresses a score after a round of discussion. This discussion of each attribute allows the participants to share their knowledge with each other. The goal of the discussion is to reach consensus, meaning that after knowledge has been shared between participants with different amounts of experience, one score is assigned on which all participants agree. The result is quantitative data in the form of scores based on personal experience and the knowledge of a group, visualized in an evaluation summary (see Figure 8.3).



	Functional suitability	Performance efficiency	Compatibility	Usability	Reliability	Security	Maintainability	Portability	Ease of implementation	Ease of learning
Group score	0	-2	1	0	1	0	3	0	4	3
Average	1,25	-0,75	0,75	0,00	1,50	-0,50	3,50	0,25	4,25	2,25

FIGURE 8.3: SPEM evaluation summary (observer pattern)

SPEM consists of four activities and three deliverables, as shown in Figure 8.2. The first activity focuses on creating participant profiles.[1] These profiles are forms containing fields for the participant's name, job description and years of experience. Additionally there are input fields for the pattern name and experience with the pattern. The participant profile also provides a list of quality attributes, sub-attributes and pattern attributes. For each item on this list, the possibility is provided to give a personal score. The evaluator introduces the method to the participants by explaining each deliverable and the focus group session protocol. In the protocol all activities and actors are listed and described. Thereafter the evaluator asks the participants to fill out part one of the participant profile.

In the second process, personal scores are assigned to an attribute. During the evaluation, the scores are recorded in the personal score list. After the evaluation, the personal scores are entered in the score table. The score table contains rows with all attributes used in the evaluation and columns containing all personal scores, average scores, standard deviations and group scores. The evaluator

[1]Templates can be found on http://www.staff.science.uu.nl/~kabbe101/PATTERNS2014/

introduces an attribute by giving a short description. The participants are then asked to assign a score to the attribute and all corresponding sub-attributes.

In the process assign group score, a group score is assigned to an attribute and noted in the score table. The group score is a score which is produced by gaining consensus, which means all participants partake in a discussion. The focus of the discussion is to exchange arguments on the score of an attribute. If consensus is reached among all participants, the resulting group score is assigned and noted on the score table. If consensus is not reached, the group score is not assigned, and no score will be noted in the score table. The evaluator initiates a discussion on the current attribute by asking a single participant's score and motivation for the score. Other participants are free to respond and exchange views, directed by the evaluator. If the discussion ends or if no time if left, the evaluator asks the participants if they have reached consensus. When consensus is reached, the group score is recorded in the score table.

When all attributes have been evaluated, an evaluation summary is created. The evaluation summary is a combination of all participant profiles and a filled out score table. Additionally, a new form is added containing the name of the evaluator, date and threats to validity. This gives the evaluator the opportunity to note any occurrences that are not expressed in the main deliverables. This process is performed by the evaluator at the end of the focus group session and concludes the evaluation.

8.5 Method Evolution

This section discusses how the initial method evolved and shows the explicit changes made to the method based on the expert evaluation sessions.

8.5.1 Initial Method Construction

Expert interviews formed the basis of the initial version of the SPEM method. Two software architects from different companies cooperated to share their views on software pattern evaluation. Understanding which attributes play a role in pattern evaluation and how they could be quantified was the goal of the interview. During the interview, a list of quality attributes derived from ISO/IEC 9126 (ISO/IEC, 2001) and ISO/IEC 25010 (ISO/IEC, 2011) was discussed, the latter being preferred by the interviewees. Although both interviews had different

results on the importance of each individual attribute of the standard, none could be excluded. Ease of learning and ease of implementation are both attributes describing characteristics of software patterns. Both attributes should be included in software pattern evaluation as they play an important part in software pattern selection.

Scenarios are often used in software architecture evaluation, but do not fit pattern evaluation. The fact that patterns are evaluated without a specific implementation in mind makes the use of scenarios irrelevant. A software architect should interpret the results of pattern evaluation by relating it to their own project. When attributes are quantified using a score, it should be possible to assign a negative value. Patterns can affect software quality in a negative way or have negative characteristics, which a score should be able to express. The range of the scores should be between a five and ten point scale. At larger ranges, it would be difficult for an architect to assign an accurate score.

When multiple architects perform a pattern evaluation, they are likely to have varying degrees of experience. Experience is key in understanding software patterns and their effect on software quality. It is important to assign a score to an attribute that takes into account the varying degrees of experience software architects have. This should be done using discussion and consensus. In a discussion, those who have more experience can share their knowledge with those who have less experience. Together working towards consensus can improve the level of knowledge of the participants and consequently improve the score. Software pattern evaluation should be performed with at least one architect who has experience using the pattern that is being evaluated. This restriction makes sure the evaluation yields a valuable result.

Based on these interview results a method was constructed incorporating the following:

- All attributes and sub-attributes from ISO/IEC 25010 (ISO/IEC, 2011).
- Two additional attributes; *ease of implementation* and *ease of learning*.
- Scoring ranging from -5 to $+5$
- Discussion after each attribute
- The goal of trying to reach consensus on each attribute

8.5.2 Method Evaluation

Using a design science approach, the initial method was evaluated and improved over several iterations. A total of three focus group sessions were hosted to evaluate the method. In these sessions, the method was carried out by evaluating a software pattern. All sessions were lead by an evaluator, who presented a pattern and queried the participants systematically on the different consequences of applying the pattern. The evaluator also encourages and guides the discussion among participants. At the end of the focus group session, participants were asked to fill out an evaluation form regarding their feedback on the method in order to enhance the external and construct validity of SPEM. The survey consisted of the following six questions:

Q1. Is the information asked on the participant profile?
Q2. Does the introduction provide enough information?
Q3. Does the introduction of attributes provide enough information?
Q4. Is the score range sufficient?
Q5. Does the score table include all relevant score data?
Q6. Do the score table and diagram enable pattern comparison?

The feedback gathered in the evaluation forms and experiences from hosting the sessions were the basis for each new iteration of the SPEM method as is typical for incremental method evaluation and improvement (Peffers, Tuunanen, Rothenberger, and Chatterjee, 2007).

First focus group evaluation — During the first focus group session, four software architects participated, each having over nine years of experience in software development. During this session, the observer pattern was evaluated using the initial version of SPEM. The pattern was selected based on the experience of the participants with the pattern. The session took approximately two hours. During the session, the quality attribute 'functional suitability' and corresponding sub-attributes appeared to be unclear to the participants. It was not possible to assign a score as the attribute demanded a specific context. Not having a description for sub-attributes was confusing and diverted discussions to the definitions of certain sub-attributes. Table 8.1 shows the responses on the feedback survey conducted after evaluation session 1.

Based on the results presented in Table 8.1, it shows that almost all experts indicated that the method can be satisfactory used for its purpose. Question 2,

Question	P1	P2	P3	P4
Q1	Yes	Yes	Yes	Yes
Q2	N/A	N/A	Yes	N/A
Q3	Yes	Yes	Yes	Yes
Q4	Yes	Yes	Yes	Yes
Q5	Yes	Yes	Yes	Yes
Q6	No	Yes	Yes	Yes

TABLE 8.1: Feedback on evaluation session 1

however, shows a high number of abstentions, which is explained by the fact most experts were already familiar with the pattern discussed during the evaluation. Because of this, it was hard for them to give any feedback about the pattern introduction. Further evaluation of the method is performed in the second focus group session. Based on the first evaluation session, the following improvements were incorporated in the method:

- **Removal of attribute 'Functional suitability'** — This attribute, including its sub-attributes turned out to be irrelevant based on the focus group session. Functional suitability can only be assessed by looking at specific implementations.
- **Including a description for all sub-attributes** — A description of each attribute, including all sub-attributes was needed. This way different interpretations of attributes can be precluded.

Second focus group evaluation — The second focus group session was performed with ten participating master students. The students have an information systems background and were all enrolled in the Software Architecture course, which prepared the students for the focus group session. The primary goal of this evaluation session is to evaluate the method, using participants with a low level of experience.

The Access Point pattern was evaluated and each time after the introduction of a quality attribute, participants were free to discuss the attribute without any intervention from the evaluator. This resulted in lengthy discussions making the session take longer than anticipated. Discussions should be halted by the evaluator after a certain period based on the time that is available. Assigning scores to sub-attributes and discussing them was time-consuming. Sub-attributes needed a less prominent role in the method. It was not always possible for participants to assign a score to an attribute. Therefore, it should be possible to have an explicit

option stating that no score is assigned, instead of leaving it empty which might imply a neutral score. Also, the introduction of the pattern allowed for ambiguous interpretations, leading to discussion and debate.

Question	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
Q1	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	No	No
Q2	Yes	No	No	Yes	No	No	No	No	No	No
Q3	No	No	No	Yes	Yes	No	No	No	No	No
Q4	Yes	No	No	N/A	N/A	Yes	No	Yes	Yes	Yes
Q5	Yes	No	N/A	Yes	Yes	Yes	Yes	Yes	Yes	No
Q6	Yes	N/A	N/A	Yes	N/A	No	Yes	Yes	Yes	Yes

TABLE 8.2: Feedback on evaluation session 2

The results of the survey conducted after the evaluation session are presented in Table 8.2. The results show a decline in satisfaction about the method. For example, according to 80% of participants, the pattern introduction did not provide enough information in order to evaluate the pattern. This result is caused by the fact that participants had no experience using the pattern, meaning they needed an extensive introduction in order to understand and evaluate it. Also, again according to 80% of the participants, the introduction of the quality attributes was not sufficient enough. The evaluation method is meant to be accessible to both experienced and inexperienced software architects, which means the method needs to be adapted. Based on the second evaluation session, the following improvements were incorporated in the method:

- **Sub-attributes removed** — Because discussion on sub-attributes took too long, they were removed from the method.
- **Added an option to give an attribute no score** — An explicit way was added for participants to indicate they do not want to give a score to a certain attribute.
- **More focus on pattern introduction** — The pattern needs to be thoroughly explaining to prevent discussions.
- **More focus on explaining what the scores represent** — Scores represent the impact the evaluated pattern has on software quality or characteristics of the pattern itself. This distinction needs to be clear in order to properly assign scores.
- **Stronger role of the evaluator** — The evaluator needs to direct the discussions. Apart from initiating discussions, they should also be halted. Time keeping is the responsibility of the evaluator.

Third focus group evaluation — The third focus group session was performed with eight software architecture students, al different from the students used in the second focus group evaluation. Although sub-attributes did not receive a score due to the changes based on the previous iteration, they were referred to in discussions to better understand an attribute. Therefore, it is important to include the sub-attributes in the method. Discussions for each sub-attribute would increase the time to complete an evaluation substantially. Personal scores were assigned to sub-attributes while discussions, and consequently group scores, related to the sub-attributes were left out.

Question	P1	P2	P3	P4	P5	P6	P7	P8
Q1	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Q2	Yes	No	Yes	Yes	Yes	Yes	No	Yes
Q3	No	Yes	No	Yes	Yes	No	Yes	Yes
Q4	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No
Q5	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes
Q6	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

TABLE 8.3: Feedback on evaluation session 3

The results presented in Table 8.3 show an improvement to the results presented in Table 8.2. Overall, we see that compared to the second focus group session, the responses to the third session are more positive. In the second session, 42% of the answers were negative, indicating that many activities and deliverables needed to be adapted. The changes to the method, evaluated in the third evaluation session, resulted in only 15% of negative answers. Based on the third evaluation session, the following improvements were incorporated in the method:

- **Sub-attributes added** — Can help the understanding of attributes and provide more detail to the data.
- **Sub-attribute discussions removed** — Gives the sub-attributes a less prominent role in the method and focusses more on attributes.
- **Descriptions for each attribute / sub-attribute added to participant profile** — Allows the participants to read descriptions of attributes independent of the evaluator.

After the three sessions, the final method (i.e. SPEM) was created.

8.6 SPEM Impementation

SPEM is created to evaluate software patterns in general, without a specific implementation in mind. This enables the option for comparison of software patterns, because each pattern has been evaluated as an abstract solution. It prevents unbalanced comparison between patterns based on different implementations. There is a trade-off between easy to compare generic evaluation and implementation specific evaluation. An implementation specific evaluation provides more accurate data, but it can only be compared to evaluated patterns based on the same implementation. A generic evaluation might not be as accurate, but ensures all evaluated patterns can be compared. SPEM can be used for implementation specific evaluation with few adjustments. It requires the evaluator to explain that the scores should be assigned with an implementation in mind. There needs to be an input field describing the implementation on the score table. With these adjustments, an evaluation session would be identical to SPEM and allows for use of all processes and deliverables used in SPEM.

This study provides knowledge on software pattern evaluation by introducing a method to evaluate software patterns. The data SPEM evaluations provide further expands the body of knowledge on patterns. It adds retrospect to the existing software pattern documentation and provides insight on the impact patterns have on software quality. A collection of SPEM evaluation results provides valuable knowledge on the understanding of software patterns and software quality. A knowledge base would enable the disclosure of SPEM evaluation results and would allow results to be combined and compared. From an industrial perspective, a SPEM knowledge base would enable software architects to share their knowledge on software patterns. It would make knowledge available to aid in software pattern selection, leading to better decision making and overall software quality. It is through sharing knowledge that software pattern selection can reach a higher level of maturity, allowing for a structured way of comparing software patterns.

SPEM uses discussion and consensus to obtain quantitative evaluation data. This method of quantification was introduced to cope with different experience levels among participants. It has imposed a constraint on the method of data gathering used in SPEM. As discussions require interaction between participants, all participants need to be able to communicate with each other at the same time. Therefore, SPEM is used in focus group sessions, limiting the number of participants. A trade-off exists between a more accurate score based on consensus with

a small number of participants and a less accurate but more reliable score with a large number of participants.

8.7 Conclusion

SPEM is a transparent software pattern evaluation method which can be used to compare patterns. It is used to evaluate relevant attributes of patterns based on the experience of software architects. SPEM provides quantitative data on attributes in the form of scores. The data can be interpreted and visualized to allow for software pattern comparison. This answers the main research question **(MRQ)**.

The question "Which attributes are relevant in pattern evaluation?" **(SQ1)** is answered with a list of attributes, consisting of quality attributes and pattern attributes. The quality attributes are based on ISO/IEC 25010 and modified for pattern evaluation, resulting in the following set of attributes: *a*) Performance efficiency, *b*) Compatibility, *c*) Usability, *d*) Reliability, *e*) Security, *f*) Maintainability, and *g*) Portability. These attributes can be quantified in a manner that allows for comparison **(SQ2)** by rating the different the attributes by experts in a focus group setting. It requires that personal scores ranging from -3 to +3 are assigned to all attributes and sub-attributes. A group score is assigned to all attributes after a discussion and reaching consensus. All scores are noted in the score table.

Conclusion

Chapter 9

Conclusion

This dissertation consists of two parts, each having a different focus and provide conclusions to two perspectives on the problem statement. The main conclusion is that variability in multi-tenant enterprise software is indispensable and can be documented in the form of software patterns. The conclusion of the second part is that software patterns are useful instruments in the architecting process and should be employed to compare different solutions and structure the decision process. The Main Research Question (MRQ) answered in this dissertation is:

> **MRQ -** *How can variability in multi-tenant enterprise software be implemented?*

The question is answered by providing a collection of software patterns (see Appendix A) that help in solving multi-tenant enterprise software design problems. The patterns describe different design problems and propose solutions, together with consequences of applying the pattern. Using the catalogue, software architects have an essential toolbox for addressing variability problems in multi-tenant enterprise software. The patterns have been gathered from focus groups and interviews with software architects during case studies at software companies. Also, the intrinsic role of patterns in the architecting process is discussed, in terms of pattern selection, comparison and evaluation. In this section, the research questions are discussed, and the evaluation of the answers presented. After the evaluation, implications of the study are discussed, followed by reflections on the research area. The main limitations of the research and future research topics related to variability in multi-tenant enterprise software are presented at the end of this section.

9.1 Contributions and Evaluations

In the dissertation, the following nine Research Questions (RQs), answers and contributions are presented:

RQ 1. *How can patterns be employed to implement variability in multi-tenant enterprise software?*

This question is answered in Part I, based on four sub questions, which are answered subsequently in Chapter 2 to 5. A summary of the answers is provided below:

RQ 1.1. *What is the concept of multi-tenancy?*

The term "Multi-Tenancy" is frequently used in academic literature and by practitioners, but no clear definition of Multi-Tenancy exists, leading to inefficient and potentially confusing communication among them. Based on the analysis of 761 research papers and 371 industrial blogs on multi-tenancy, which have been identified using a Systematic Mapping Study (SMS), the following definition of multi-tenancy is formulated:
"Multi-tenancy is a property of a system where multiple customers, so-called tenants, have the possibility to configure the system; it allows them to transparently share the system's services, applications, databases, or hardware resources, with the aim of lowering costs".

Multi-tenancy is also characterized as a research domain which is still evolving and of which no clear research direction exists. Chapter 2 identifies seven research themes that can steer the multi-tenancy domain, which are:

- **Quality Assurance** — An investigation into how customization of the multi-tenant application affects quality (e.g. performance). For example, can one general SLA be enforced, or should each tenant get a tentant-specific SLA?
- **Industry Validation** — With industrial multi-tenant solutions being developed right now, the next step for researchers is to work closely together with industry to validate research ideas on actual multi-tenant software systems.
- **Balancing & Placement** — Research on the opportunities to develop better load balancing algorithms, taking into account the historical usage of the application by the different tenants, needs to be performed.

- **Database** — Investigation into how to isolate data in a secure way. Additionally, the partitioning of data, based on tenant interaction should be addressed.
- **Platform Development** — There should be an open platform available for multi-tenant applications. Researchers and industry should work together in achieving and maintaining such a platform.
- **Security** — The topic of security should be inextricably related to both research and development of multi-tenant software. Security should play an even more indispensable role in multi-tenant systems than it already does in multi-instance and multi-user systems since data of all tenants can be stored in the same database or table.
- **Variability** — There should be more awareness of the importance of variability in multi-tenant software. Variability should be inextricably tied to the concept of multi-tenancy in implementation and communication.

These themes are also a call to other researchers for future work on multi-tenancy and are discussed in more detail in Chapter 2.

RQ 1.2. *How are software patterns used to implement variability?*
Software patterns have proven to be a useful instrument for documenting common solutions to frequently occurring problems in software engineering. Patterns can contain many different elements, of which the following are used in this research:

- **Context** — Sets the stage where the pattern takes place.
- **Problem** — Explains what the actual problem is.
- **Forces** — Describe why the problem is difficult to solve.
- **Solution** — Explains the solution in detail.
- **Consequences** — Demonstrates what happens when you apply the solution.

The pattern elements can be mapped to variability related attributes as shown in Figure 9.1.

The use of patterns to document variability problems compels to study forces and consequences of a solution in a structured way, enhancing the rigour of developing the solution. By using appropriate patterns, software architects and decision makers can implement variability in multi-tenant enterprise applications in a predictable way. The structured nature of patterns enhances the predictability by providing clear

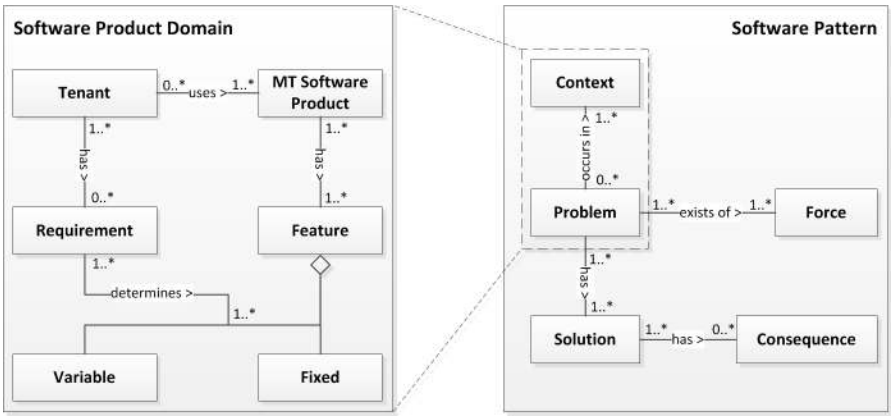


FIGURE 9.1: The role of variability patterns in multi-tenant enterprise software

descriptions of defined elements of a complete solution. Software patterns are a valuable tool in the design of multi-tenant applications, as can be seen in Chapter 3.

RQ 1.3. *What are the trade-offs of providing more variation in multi-tenant enterprise software?*

When the number of different, or opposing, customer requirements for a software product grows, the need for variability in the solution grows. A software product line implementation satisfies a high need for variability but fails to suffice if the number of customers grows. The specific software solution most suited for implementing an appropriate level of variability also depends on the need for resource sharing. Figure 9.2 shows how multi-tenancy plays an important role implementing variability of the needs for resource sharing are high.

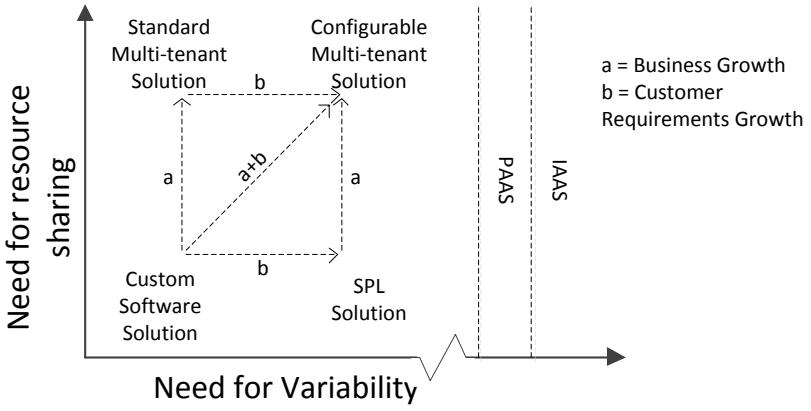


FIGURE 9.2: Level of variability versus Number of users

Software patterns help to implement variability to different degrees in multi-tenant enterprise software. The following three patterns are identified in Chapter 4:

- **Customizable Data Views Pattern** — Gives the tenant the ability to indicate and save his preferences on the representation of data shown.
- **Module Dependent Menu Pattern** — Provides a custom menu to all tenants, only containing links to the functionality relevant to the tenant.
- **Pre/Post Update Hooks** — Provides the possibility for tenants to have custom functionality just before or after an event.

Using the patterns, architects and developers are supported in implementing variability in configurable multi-tenant solutions.

RQ 1.4. *How does the CQRS pattern influence the variability of a software product?*

The CQRS pattern dictates the strict separation between commands and queries in the entire software product, which causes software products to be more scalable and variable than traditional multi-tier based products. The scalability and variability offered by CQRS are crucial in deploying multi-tenant enterprise software. An overview of CQRS is depicted in Figure 9.3.

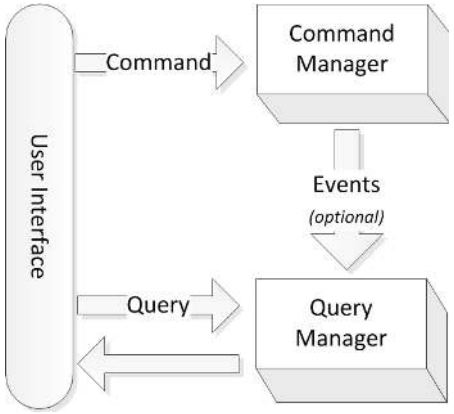


FIGURE 9.3: Overview of the CQRS pattern

A case study was performed, at a software company using the CQRS pattern. During the case study, the software architecture of the relevant product was examined, based on expert interviews. The following seven sub-patterns have been identified, that can be used to implement CQRS:

- **Event Sourcing** — Different events are broadcasted by the command manager to be processed by different components
- **Event Store** — Central place in which the events are stored and changes can be reconstructed from.
- **Aggregate Root** — Storing and processing all properties and entities that are dependent on each other together
- **Command Handler** — System capable of catching one or more commands and passing it through to an object capable of performing the command.
- **Query Model Builder** — Listens to events coming in through the event bus, and create a view of the data needed by the query manager.
- **Query Handler** — Receives all queries and checks the query store for views created by the Query Model Builder.
- **Snapshotting** — Storing the state of the aggregate root, together with the events, every n^{th} event.

None of the sub-patterns are obligatory for the implementation of CQRS and can be used in any combination. In CQRS the *commands* and *queries* are separated. Because of this separation, software application using the pattern, with a combination of sub-patterns, can easily implement variability. More details on the CQRS pattern and the sub-patterns can be found in Chapter 5.

RQ 2. *How can software patterns become an intrinsic part of the architecting process?*

This question is answered in Part II, based on three sub-questions, which are answered subsequently in Chapter 6 to 8. A summary of the answers can be found below:

RQ 2.1. *How can software architects be supported in the selection process of choosing an applicable multi-tenant architecture pattern?*

Twelve Multi-Tenant Architecture (MTA) patterns are constructed that can be used to design multi-tenant enterprise software. All patterns can be defined in the tuple:

$$MTA = \langle \{AD, AS, AI\}, \{DD, DS, DB, DC\} \rangle \tag{9.1}$$

In each pattern, resources are shared on different levels related to the Application (A) and to the Database (D). The three different application levels are:

- **AD** - A Dedicated Application server is running for each tenant, and therefore, each tenant receives a dedicated application instance.
- **AS** - A single Application Server is running for multiple tenants, and each tenant receives a dedicated application instance.
- **AI** - A single application server is running for multiple tenants, and a single Application Instance is running for multiple tenants.

Concerning the database, resources can be shared on the following four levels:

- **DD** - A Dedicated Database server is running for each tenant, and therefore, each tenant receives a dedicated database.
- **DS** - A single Database Server is running for multiple tenants, and each tenant receives a dedicated database.
- **DB** - A single DataBase server is running for multiple tenants, data from multiple tenants is stored in a single database, but each tenant receives a dedicated set of tables.
- **DC** - A single database server is running for multiple tenants, data from multiple tenants is stored in a single database and a single set of tables, sharing the same Database sChema.

Based on the discussed levels of multi-tenancy, a full set of multi-tenant architecture patterns is created.

Choosing the appropriate pattern is a challenging task, depending on many different factors. During the selection process, architects have to take the effect of a specific pattern on many different, sometimes contradicting, quality attributes into account. Multi-tenant Architecture Assessment Model (MAAM) supports architects and decision makers in selecting applicable multi-tenant architecture patterns by focusing on consequences of applying the patterns. By using MAAM, decision makers are forced to focus on a limited set of consequences they consider important, helping them structure their decision. A collection of rules of thumb (e.g. Focus on the database dimension) is presented to give additional guidance in the decision process. Chapter 6 provides more elaboration on the MTA patterns and assessment method.

RQ 2.2. *What are the influences of variability patterns on software quality attributes?*

Different ways of implementing variability in online software products exist. It is unclear, however, what the consequences are of specific solutions and what patterns are preferred in certain situations. In order to implement variable functionality and a variable data model in multi-tenant enterprise software, four software patterns are presented. Firstly, to dynamically adapt functionality in online software products, the following to patterns can be used:

- **Component Interceptor Pattern** — A single application server, in which the interceptors are tightly integrated with the application, because they run in-line with normal application code. This pattern serves best for adapting functionality of small projects.
- **Event Distribution Pattern** — The application generates events at extension points, which are distributed by a broker. At each extension point, the standard component is programmed to send an event indicating the point and appropriate contextual data to a broker. This pattern is best for large projects.

To extend the data model of an application, the following two patterns are identified:

- **Datasource Router Pattern** — The application uses a different database instance (or schema) for each tenant. Custom properties are then added to the database as normal fields.
- **Custom Property Object Pattern** — Data from all tenants is stored in a single database with a single schema. Any additional data like custom properties is modeled in the design of the application as separate custom property objects which are stored in the existing static schema. Because all data is stored in a single database, components using that data need to be aware of multi-tenancy and explicitly query for data of a specific tenant.

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. An elaborate comparison between all patterns can be found in Chapter 7.

RQ 2.3. *How can software patterns be objectively evaluated and compared during the enterprise software architecting process?*
Using the Software Pattern Evaluation Method (SPEM), software architects can evaluate patterns based on different quality attributes. The

evaluation is used to compare which patterns are most fitting for their architecting decisions.

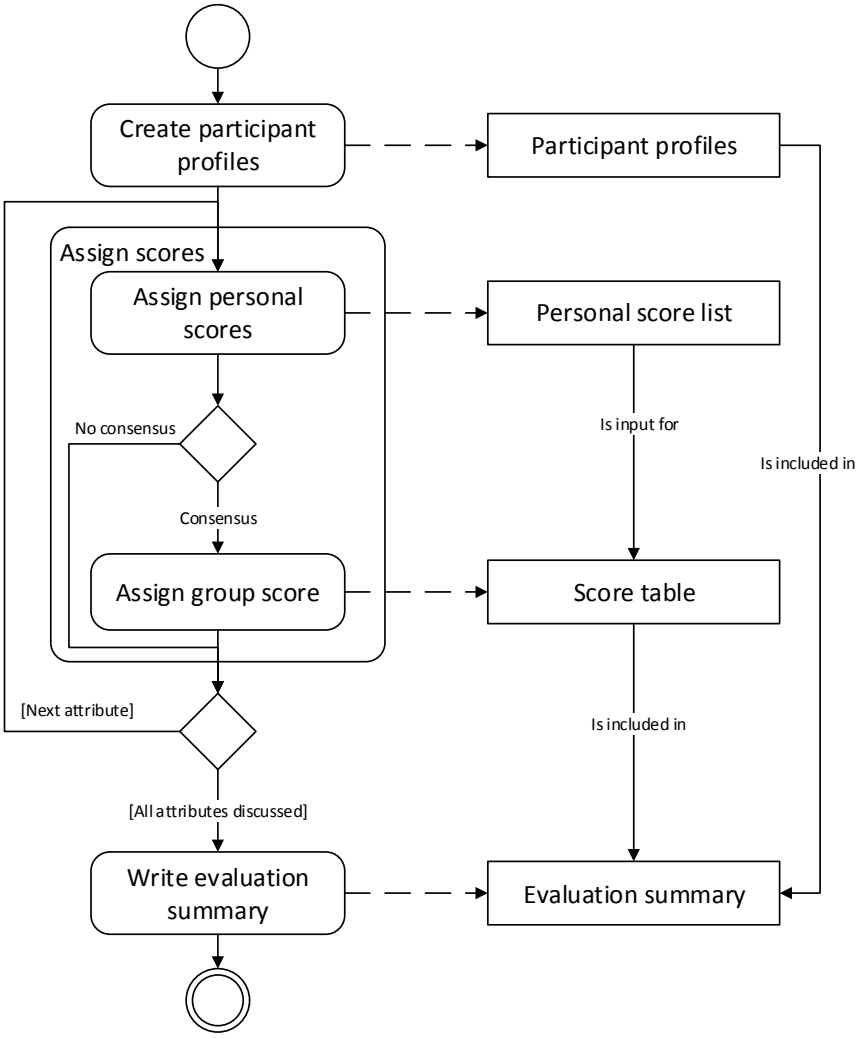


FIGURE 9.4: SPEM: Software Pattern Evaluation Method

Figure 9.4 shows an overview of the method, which takes place in a focus group setting. First, profiles are created of all participants of the focus group. After this, a pattern is presented followed by a sequence of quality attributes. When a quality attribute is presented, participants fill in their personal score concerning the expected influence of the pattern on the attribute. A discussion follows in which the group tries to agree on a group score for the attribute. After all quality attributes are discussed, a pattern evaluation report is created. More details on the method construction and how to perform the method can be found in Chapter 8. Using the Software Pattern Evaluation Method (SPEM), architects can

make better-informed architecting decisions, saving them time, effort and evolution challenges in the future.

9.2 Implications

The result of this research has different implications for the software engineering and software architecture research community, and the software industry. The largest implications are in the following three areas:

- Providing a unambiguous view on multi-tenancy
- Giving a practical set of multi-tenant architecture patterns
- Presenting a method for the evaluation of software patterns (SPEM)

All of the areas are discussed below.

Unambiguous view on multi-tenancy — Considering the apparent unstoppable shift towards the SaaS deployment model and cloud computing paradigm that can be observed in the software industry, sharing resources among many customers becomes of undeniable importance. In order to streamline research on this topic of resource sharing and foster adoption of academic results by industry, a common vocabulary is indispensable. Multi-tenancy is a concept which is concerned with the sharing of software resources among different customers, but which is used differently between and among academia and industry. There is no consensus on what elements of a software product need to be shared among tenants in order to make a product multi-tenant. Also, the number of tenants that can be catered for by a software product is not clear. The many varying uses of the term can not only confuse software vendors, but can even be harmful when it comes to buying the appropriate third party software. Every vendor can employ another interpretation of multi-tenancy, which means a multi-tenant product offering may not be able to share the resources or offer the variability expected by the client.

We now have postulated an unambiguous definition of multi-tenancy in Chapter 2, aiding the software industry and software engineering research field, by giving a common way of talking about multi-tenancy, so confusion can be avoided. The definition, together with the multi-tenancy research agenda, also helps researchers in structuring future research in the area multi-tenancy and SaaS, leading to a more mature research area. The view on multi-tenancy presented in this dissertation

implicates a common ground among research and industry, catalyzing multi-tenant enterprise software manufacturing.

Essential set of multi-tenant architecture patterns — Defining a suitable architecture for a software product is crucial. Especially when an application is hosted online and may serve up to millions of customers and users, thorough knowledge on the consequences of architectural design decisions is inevitable. Many enterprise software producing companies struggle to select the appropriate architecture for their online software product. Quality attributes, such as maintainability, scalability or variability, are important factors in the architecting process, which are often hard, or almost impossible to assess beforehand.

The twelve Multi-Tenant Architectures (MTAs), as explained in Chapter 6 and presented in Appendix A, help software architects and decision makers in structuring the architecting process. Before decisions are made, the consequences can be assessed by looking at the evaluations of the MTAs. Communication for both industry and academia is improved by using the set of MTAs, because they provide clear patterns that can be referred to. The patterns provided in Chapter 6 entail a set of multi-tenant architecture patterns for enterprise software. This set provides a foundation for academics to efficiently visualize and communicate about multi-tenant architectures and did not exist before. The patterns form the basis for future research and enable more in-depth analysis of the quality consequences of using a specific multi-tenant architecture. A much-needed essential set of patterns for multi-tenant architecting is provided by the results of this dissertation. Replication of the study would provide the same set of essential architecture patterns, possibly extended by hybrid patterns, based on the patterns provided in this research.

Software pattern evaluation method — For years, software patterns have been used to structure and document design solutions in software engineering. Numerous patterns exist and are capable of solving a wide variety of different, and similar, problems. Patterns often include a description on the consequences of applying the pattern, but these consequences are seldom presented in quantitative form and focus on only a selection of relevant criteria. Analyzing how the implementation of a software pattern affects the quality of a system is a domain that is still largely uncharted.

With the creation of SPEM, a proof of concept is provided for the evaluation of patterns, related to their expected impact on the system implementation. One goal of SPEM is to create more awareness for this kind of evaluations. Our research

paves the road for pattern and software architecture researchers to study the evaluation of software patterns in more depth. Using the knowledge of architects to assess the consequences of applying a specific pattern, by means of focus groups, is a new approach for software quality evaluation. In current literature, measures exist for evaluating the quality of a software architecture. Also, efforts have been done to measure the effects of a pattern on a specific system implementation. A combination of both, in which the effect of a pattern on a software architecture is assessed, has not been performed before. The results of this dissertation fill this gap and give patterns a prominent role in the entire architecting process.

9.3 Reflection

This section gives an overview of reflections on the research project in general, but also more specific on the research process, methods used and observations on the software industry. The following four reflections will be discussed:

- Software patterns are underused in the Dutch software industry.
- It is impossible to individually measure pattern consequences using a test environment.
- Case studies are invaluable in software pattern research.
- On the future of patterns.

Unawareness of patterns in the software industry — Software patterns have been used in this project to document design solutions, but also serve as means of communication and comparison tool in the architecting process. The primary method used to gather the patterns is by performing case studies at Dutch software producing companies. During the interviews that were performed with software architects and other experts at the companies, the knowledge on patterns was often limited. Many interviewees see the 23 design patterns, as proposed by Gamma, Helm, Johnson, and Vlissides (1995) (e.g. Observer or Visitor patterns), not only as the seminal patterns, but often these patterns are the only patterns familiar to them. We observed a clear unawareness of software architects on the potential of software patterns. Many have the feeling they were constantly 'reinventing the wheel', but do not know the potential of patterns to document, communicate and compare solutions. Also, many express the urge for a structured way of documenting solutions and ways of comparison.

Software patterns can offer the needed structure but are unacknowledged and underused in the Dutch software industry. Based on our observations and personal reflections, patterns are not used enough in the software industry. In my opinion, the software industry would benefit greatly by making a more explicit use of patterns. A potential inhibitor is **(1)** the lack of pattern knowledge infrastructures in place. Additionally, **(2)** patterns are not used sufficiently in documentation within software companies. The root of this problem may be in the **(3)** conventional and limited coverage of software patterns at universities; inadequate pattern education. Lastly, responsibility for proper pattern use throughout the entire development cycle, primarily needs to be carried by the software industry.

Envisioned test environment for patterns — An important part of this research is the assessment of the consequences of applying a pattern on the quality of a software product. At the beginning of the research, the aim was to measure the differences between software patterns by creating a test environment in which different patterns could be set up. Each set-up could then be tested on different quality attributes, depending on several scenarios. The idea of creating a test environment is based on current practices of performance measuring. The problem with measuring the results of applying a specific pattern, is that we measure the *implementation*, instead of the *pattern*. The problem lies in what defines a pattern. In describing a pattern, always a generic solution is given, instead of a specific implementation. This means an architect or developer always has to interpret the proposed solution and implement as he finds suited. Because of this abstract nature of patterns, it is impossible to measure the effect of a pattern in a test environment. One always measures the specific set-up in the test environment, instead of the pattern itself.

In this research, the test environment is replaced by expert interviews and focus groups to collect data about the consequences of applying specific patterns. This leads to a more indirect way of measuring different consequences of applying patterns, but combines the experience and expertise of many different professionals. By gathering this knowledge in a structured way, we get valuable results and evaluate many different interpretations of a pattern at once. Combining the knowledge of all architects, allows us to evaluate many implementations simultaneously, instead of only one implementation. When a future, central a collaborative pattern catalogue is in place, however, many researchers could test similar patterns in a test environment. The collaborative testing of patterns could generate a large enough data set to distil knowledge on the *pattern*, instead of the specific *implementation*.

Value of case studies in pattern research — Case study research is sometimes criticized for being very case specific and lacking generalizability. During this research, an appropriate research method had to be selected in order to identify the different software patterns. An often used rule of thumb in pattern gathering is that a solution should be observed in at least three occasions before it can be rightfully documented as a pattern. The best way to do this is by performing case studies at different software producing companies, in which software architects are interviewed on how they solve specific problems in their software product. Using source code analysis to identify potential patterns will not work for architectural patterns and other high-level solutions. Also, the documentation available at the case companies often proved to be too poor to study for the sake of pattern identification.

By using interviews in a multiple case study research design, as discussed in Section 1.4.2, we were able to gather different patterns efficiently. The claim that case study research lacks generalizability is not true for pattern gathering if clear problems are analyzed. In fact, because patterns are supposed to be proven common practices, case study research is a well-fitting method for identification.

Future of patterns — Patterns can be used as an educational tool to train software architects. Architects are human beings that solve problems by weighing alternatives and making various decisions. These decisions are anchored in their education, their experience, and the problem at hand. Patterns enable architects to make informed decisions, without having to trust on gut feeling and experience alone. This research, for instance, provides software architects with a decision tool in chapter 6 for choosing appropriate multi-tenant deployment architectures for online applications, supporting architects in decision making. As patterns are becoming an increasing part of common software engineering practice, tools will increasingly support these building blocks. Similar to three-dimensional drawing tools for building architects that contain basic constructs such as walls, windows, and stairs, we expect integrated development environments to assist in rapid deployment of reference implementations of (for instance) the model-view-controller pattern, the CQRS pattern, and many others. We can even imagine that refactorings take on the shape of refactorings towards a specific type of pattern to optimally support developers in re-engineering their software.

One often heard criticism among developers is that design patterns are essentially solid object-oriented practices. Although this is true for some of the lower level software patterns (i.e. idioms and some design patterns), we must strongly object

to a generalization to all patterns. The patterns in this work and presented by many others are advanced constructs that solve larger scale design problems in specific problem domains. These constructs are typically complex and provide a blueprint solution for a problem that is larger than just its programmatic implementation. Another often heard criticism is that software patterns are essentially constructs missing in the language that thus had to be defined by outsiders. Effectively, this comment suffers from the same problem as the previous comment, which is that some patterns are much more complex than can be fixed by one construct in a language. It is important to keep in mind that different levels of patterns exist, as discussed and explained in section 1.3.5, and comments about one specific type of patterns does not need to apply to all patterns. Patterns should be used as reference solutions. They should not kill developer creativity and should not turn them into "macro-composers".

9.4 Limitations and Future Research

The limitations of the results, presented in this dissertation, are discussed in this section, followed by directions on future research.

Variability in the functionality or data model of multi-tenant enterprise software can be implemented in numerous ways, each having their specific consequences. This dissertation describes and compares a number of patterns to implement variability, but the collection provided is not all-embracing yet. More patterns need to be identified and evaluated in order to get a complete view on the architecting options in multi-tenant enterprise software. Also, the patterns need to be stored in a central and open catalogue, to enable easy access to the patterns for software architect and researchers. A central, online, catalogue will also enable distributed identification and comparison of patterns and pattern implementation consequences. A risk of this research is the intensive use of patterns in documenting, evaluating and comparing design solutions. While this is done deliberately, it could harm the acceptance of the results. Software patterns are not universally used by academics and industry, leading to a potential hesitation in the adoption of the results of this research.

Researchers can only perform a limited number of cases and can never cover the entire population (i.e. all organisations producing multi-tenant enterprise software). To counteract this, we carefully selected our case companies to reflect and represent the industry as good as possible. Within our selection are three of the largest

ERP vendors in the Netherlands, together with a large group of smaller organisations, all producing enterprise software in varying domains. The results of our studies were later evaluated by experts to increase the level of generalizability. One threat is that all companies are based in the Netherlands. Because of the international character of many of the companies, however, we have no reason to think this will harm the generalizability of our results. Additionally, patterns are proven solutions to a common problem, based on multiple observations of the solution. Because of this, increasing the number of case companies would not increase the generalizability of the identified patterns. It would, however, potentially increase the number of patterns found.

Validation of research results is crucial throughout the entire research process. In this dissertation, all results are validated by discussing them in expert interviews, focus groups, or surveys. Additionally, the validity of the results is ensured by rigorously following the case study protocol that is set up beforehand and employing design science approach in which the results are constantly validated during each iteration of the cycle. Although we tried to ensure the validity of our results, a thorough, overall validation of all patterns at the same time, using a similar validation approach for every pattern is still lacking. A future, overall validation, in addition to the current disperse validation methods, could improve the validity of the current results even more.

For future research, the concept of multi-tenancy in enterprise software should be researched more extensively on the following topics:

- **Quality assurance** — An investigation into how customization of the multi-tenant application affects quality, e.g. in terms of performance. Can one general SLA be upheld, or should each tenant get a tenant-specific SLA?
- **Industry Validation** — With industrial multi-tenant solutions being developed right now, the next step for researchers is to work closely together with industry to validate research ideas on actual multi-tenant software systems.
- **Balancing and Placement** — There might be opportunities to develop better load balancing algorithms that take into account the historical usage of the application by the different tenants. Specifically, the load balancing can be targeted at looking at the different time zones in which the tenants are operating.
- **Database sharing** — A major point of concern that we noted in the blog posts is data isolation, i.e., making sure that the data of individual tenants

is shielded for other tenants. As such, an investigation into how to isolate and partition the data is a logical next step. Additionally, developing tests to make sure that data isolation is working correctly is also an interesting avenue for future work.

Performing more in-depth studies in these areas would lead to a more mature research domain and a more extensively defined concept of multi-tenancy. The current discomposure on the characterization of multi-tenancy underlines the need for further research in this area.

Regarding the patterns that are presented in this dissertation, more evaluation needs to be done to assess the effect of all patterns on software quality attributes, such as maintainability, scalability and performance. Also, more patterns can be identified if additional case studies are performed. Doing this would enrich the pattern catalogue and give software architects a wider variety of patterns to choose from. Assuming an appropriate evaluation method is used during assessment, a richer catalogue would improve the architecting process. Currently the pattern language body of knowledge is already constantly growing with recent additions such as the patterns in this thesis, but also, among others, the recent pattern collection books Cloud Design Patterns (Homer, Sharp, Brader, Masashi, and Trent, 2014) and Designing Distributed Control Systems: A Pattern Language Approach (Eloranta, Koskinen, Leppanen, and Reijonen, 2014). Little effort has gone into collecting, mapping, and classifying all software design patterns. We strongly believe this calls for an indexed pattern encyclopedia, in which patterns can be collected. Ideally, such an encyclopedia allows for annotations to be made that add knowledge on the patterns in practice, pattern variations, pattern complements, reference implementations, etc. The patterns elaborated on in this thesis could fill the section on variability in online multi-tenant software.

Bibliography

Abdullin, R. (2010). *Theory of CQRS Command Handlers: Sagas, ARs and Event Subscriptions*. http://abdullin.com/journal/2010/9/26/theory-of-cqrs-command-handlers-sagas-ars-and-event-subscrip.html.

Abowd, G., L. Bass, P. Clements, R. Kazman, and L. Northrop (1997). *Recommended Best Industrial Practice for Software Architecture Evaluation*. Tech. rep. DTIC Document.

Alexander, C., S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel (1977). *A pattern language: Towns, Buildings, Construction*. Oxford University Press, Oxford, UK.

Anjum, M. and D. Budgen (2012). "A mapping study of the definitions used for Service Oriented Architecture". In: *Prodceeings of the International Conference on Evaluation & Assessment in Software Engineering (EASE)*. IET, pp. 57–61.

Arlitt, M., D. Krishnamurthy, and J. Rolia (2001). "Characterizing the scalability of a large web-based shopping system". In: *ACM Transactions on Internet Technology* 1.1, pp. 44–69.

Armbrust, M., A. Fox, R. Griffith, et al. (2010). "A view of cloud computing". In: *Communications of the ACM* 53.4, pp. 50–58.

Arya, P., V. Venkatesakumar, and S. Palaniswami (2010). "Configurability in SaaS for an electronic contract management application". In: *Proceedings of the International Conference on Networking, VLSI and Signal Processing*. ACM, pp. 210–216.

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

Azeez, A., S. Perera, D. Gamage, R. Linton, P. Siriwardana, D. Leelaratne, S. Weerawarana, and P. Fremantle (2010). "Multi-tenant SOA middleware for cloud computing". In: *Proceedings of the International Conference on Cloud Computing (CLOUD)*. IEEE, pp. 458–465.

Babar, M. A. and I. Gorton (2007). "A tool for managing software architecture knowledge". In: *Proceedings of Workshop on Sharing and Reusing Architectural Knowledge (SHARK)*. IEEE, pp. 11–17.

Babar, M. A., L. Zhu, and R. Jeffery (2004). "A framework for classifying and comparing software architecture evaluation methods". In: *Proceedings of the Australian Software Engineering Conference*. IEEE, pp. 309–318.

Bass, L., P. Clements, and R. Kazman (2013). *Software Architecture in Practice*. Addison Wesley, Boston.

Bayer, J., . Gerard, O. Haugen, J. Mansell, B. Møller-Pedersen, J. Oldevik, P. Tessier, J. Thibault, and T. Widen (2006). "Consolidated Product Line Variability Modeling". In: *Software Product Lines*. Springer, pp. 195–241.

Beck, K., R. Crocker, G. Meszaros, J. Vlissides, J. O. Coplien, L. Dominick, and F. Paulisch (1996). "Industrial experience with design patterns". In: *Proceedings of the International Conference on Software engineering*. IEEE, pp. 103–114.

Benlian, A. and T. Hess (2011). "Opportunities and risks of software-as-a-service: Findings from a survey of IT executives". In: *Decision Support Systems* 52.1, pp. 232–246.

Bennett, K., P. Layzell, D. Budgen, P. Brereton, L. Macaulay, and M. Munro (2000). "Service-based software: the future for flexible software". In: *Proceedings of the Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, pp. 214–221.

Berg, B. L. and H. Lune (2004). *Qualitative research methods for the social sciences*. Pearson, Boston.

Betts, D., J. Dominguez, G. Melnik, F. Simonazzi, and M. Subramanian (2013). *Exploring CQRS and Event Sourcing: A journey into high scalability, availability, and maintainability with Windows Azure*. Microsoft patterns & practices, Redmond.

Bezemer, C. and A. Zaidman (2010). "Multi-tenant SaaS applications: maintenance dream or nightmare?" In: *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*. ACM, pp. 88–92.

Bezemer, C., A. Zaidman, B. Platzbeecker, T. Hurkmans, and A. Hart (2010). "Enabling multi-tenancy: An industrial experience report". In: *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE.

Bhardwaj, S., L. Jain, and S. Jain (2010). "Cloud computing: A study of infrastructure as a service (IAAS)". In: *International Journal of engineering and information Technology* 2.1, pp. 60–63.

Bondi, A. B. (2000). "Characteristics of scalability and their impact on performance". In: *Proceedings of the International workshop on Software and Performance (WOSP)*. ACM, pp. 195–203.

Booch, G. (2005). "On creating a handbook of software architecture". In: *Proceedings of the Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pp. 8–8.

Brereton, P., B. A. Kitchenham, D. Budgen, M. Turner, and M. Khalil (2007). "Lessons from applying the systematic literature review process within the software engineering domain". In: *Journal of Systems and Software* 80.4, pp. 571–583.

Brown, C. and I. Vessey (2003). "Managing the next wave of enterprise systems: leveraging lessons from ERP". In: *MIS Quarterly Executive* 2.1, pp. 45–57.

Brown, W., R. Malveau, and T. Mowbray (1998). *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, New York.

Brownsword, L., T. Oberndorf, and C. A. Sledge (2000). "Developing new processes for COTS-based systems". In: *IEEE Software* 17.4, pp. 48–55.

Budgen, D., M. Turner, P. Brereton, and B. Kitchenham (2008). "Using mapping studies in software engineering". In: *Proceedings of the Annual Meeting of the Psychology of Programming Interest Group*. PPIG, pp. 195–204.

Buschmann, F., K. Henney, and D. Schmidt (2007a). *Pattern-oriented software architecture: On patterns and pattern languages*. John Wiley & Sons, New York.

– (2007b). *Pattern-Oriented Software Architecture, Volume 4: Pattern Language for Distributed Computing*. John Wiley & Sons, New York.

Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal (1996). *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons, New York.

Buschmann, F., K. Henney, and D. C. Schmidt (2007c). "Past, present, and future trends in software patterns". In: *IEEE Software* 24.4, pp. 31–37.

Carpenter, B., G. Fox, S. Ko, and S. Lim (1999). "Object serialization for marshalling data in a Java interface to MPI". In: *Proceedings of the Conference on Java Grande*. ACM, pp. 66–71.

Chong, F. and G. Carraro (2006). *Architecture strategies for catching the long tail*. Tech. rep. MSDN Library, Microsoft Corporation.

Chong, F., G. Carraro, and R. Wolter (2006). *Multi-tenant data architecture*. Tech. rep. MSDN Library, Microsoft Corporation.

Clements, P., D. Garlan, L. Bass, J. Stafford, R. Nord, J. Ivers, and R. Little (2002). *Documenting software architectures: views and beyond*. Pearson Education, Upper Sadle River.

Cooper, H. (1998). *Synthesizing research: A guide for literature reviews*. SAGE Publications, London, UK.

Copi, I. M. and R. W. Miller (1972). *Introduction to Logic: Study Guide*. Macmillan, London, UK.

Coplien, J. O. and C. Alexander (1996). *A word on Software patterns*. SIGS, New York.

Cronbach, L. J. and P. E. Meehl (1955). "Construct validity in psychological tests." In: *Psychological bulletin* 52.4, pp. 281–302.

Cusumano, M. A. (2004). *The Business of Software*. Free Press, New York.

Dahan, U. (2010). *Clarified CQRS*. http://www.udidahan.com/2009/12/09/clarified-cqrs/.

Dillon, T., C. Wu, and E. Chang (2010). "Cloud computing: Issues and challenges". In: *Proceedings of the International Conference on Advanced Information Networking and Applications (AINA)*. IEEE, pp. 27–33.

D'souza, A., J. Kabbedijk, D. Seo, S. Jansen, and S. Brinkkemper (2012). "Software-as-a-Service: Implications for Business and Technology in Product Software Companies". In: *Proceedings of the Pacific Asia Conference on Information Systems (PACIS)*. AIS, Paper 140.

Du, J., X. Gu, and D. S. Reeves (2010). "Highly available component sharing in large-scale multi-tenant cloud systems". In: *Proceedings of the International Symposium on High Performance Distributed Computing*. ACM, pp. 85–94.

Dubey, A. and D. Wagle (2007). "Delivering software as a service". In: *The McKinsey Quarterly* 6, pp. 1–12.

Eckerson, W. (1995). "Three Tier Client/Server Architectures: Achieving Scalability, Performance, and Efficiency in Client/Server Applications". In: *Open Information Systems* 3.20, pp. 46–50.

Eisenhardt, K. M. (1989). "Building theories from case study research". In: *Academy of management review* 14.4, pp. 532–550.

Eloranta, V., J. Koskinen, M. Leppanen, and V. Reijonen (2014). *Designing Distributed Control Systems: A Pattern Language Approach*. John Wiley & Sons, New York.

Engelstätter, B. (2012). "It is not all about performance gains–enterprise software and innovations". In: *Economics of Innovation and New Technology* 21.3, pp. 223–245.

Esfahani, N., S. Malek, and K. Razavi (2013). "GuideArch: guiding the exploration of architectural solution space under uncertainty". In: *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, pp. 43–52.

Esfahani, N., K. Razavi, and S. Malek (2012). "Dealing with uncertainty in early software architecture". In: *Proceedings of the International Symposium on the Foundations of Software Engineering*. ACM, pp. 21–24.

Evans, E. (2004). *Domain-driven design: tackling complexity in the heart of software*. Addison Wesley, Boston.

Evitts, P. and D. Hinchcliffe (2000). *A UML pattern language*. Vol. 201. Macmillan Technical Publishing, London, UK.

Fink, A. (2013). *Conducting research literature reviews*. Sage Publications, London, UK.

Fisher, S. (2007). "The architecture of the apex platform, salesforce.com's platform for building on-demand applications". In: *Proceedings of the International Conference on Software Engineering-Companion (ICSE)*. IEEE, p. 3.

Flyvbjerg, B. (2006). "Five misunderstandings about case-study research". In: *Qualitative inquiry* 12.2, pp. 219–245.

Forbes (2014). *Why Cloud ERP Adoption Is Faster Than Gartner Predicts*. http://www.forbes.com/sites/louiscolumbus/2014/02/07/why-cloud-erp-adoption-is-faster-than-gartner-predicts/.

Forrester (2012). *Thoughts On The ERP Market As 2012 Shifts Into 2013*. http://blogs.forrester.com/china_martens/12-12-10-thoughts_on_the_erp_market_as_2012_shifts_into_2013.

– (2013). *Cloud Computing Predictions for 2014: Cloud Joins the Formal IT Portfolio*. http://blogs.forrester.com/james_staten/13-12-04-cloud_computing_predictions_for_2014_cloud_joins_the_formal_it_portfolio.

Fowler, M. (1997). *Analysis Patterns: reusable object models*. Addison-Wesley, Boston.

– (2003). *Patterns of enterprise application architecture*. Addison-Wesley Professional.

Galster, M. and P. Avgeriou (2011). "The notion of variability in software architecture: results from a preliminary exploratory study". In: *Proceedings of the Workshop on Variability Modeling of Software-Intensive Systems*. ACM, pp. 59–67.

Galster, M., T. Männistö, D. Weyns, and P. Avgeriou (2014). "Variability in software architecture: the road ahead". In: *ACM SIGSOFT Software Engineering Notes* 39.4, pp. 33–34.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides (1995). *Design patterns: elements of reusable object-oriented software*. Addison-wesley Reading.

Gilbert, S. and N. Lynch (2002). "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services". In: *ACM SIGACT News* 33.2, pp. 51–59.

Gold, N., A. Mohan, C. Knight, and M. Munro (2004). "Understanding service-oriented software". In: *IEEE Software* 21.2, pp. 71–77.

Goodwin, L. D. and N. L. Leech (2003). "The Meaning of Validity in the New Standards for Educational and Psychological Testing: Implications for Measurement Courses." In: *Measurement and evaluation in Counseling and Development* 36.3, pp. 181–192.

Gray, J. and L. Lamport (2006). "Consensus on transaction commit". In: *ACM Transactions on Database Systems (TODS)* 31.1, pp. 133–160.

Guo, C., W. Sun, Y. Huang, Z. Wang, and B. Gao (2007). "A framework for native multi-tenancy application development and management". In: *Proceedings of the International Conference on E-Commerce Technology and the International Conference on Enterprise Computing, E-Commerce and E-Services*. IEEE, pp. 551–558.

Hendricks, K. B., V. R. Singhal, and J. K. Stratman (2007). "The impact of enterprise systems on corporate performance: A study of ERP, SCM, and CRM system implementations". In: *Journal of Operations Management* 25.1, pp. 65–82.

Hevner, A. R., S. T. March, J. Park, and S. Ram (2004). "Design Science in Information Systems Research". In: *MIS Quarterly* 28.1, pp. 75–105.

Hevner, A. and S. Chatterjee (2010). *Design research in information systems: theory and practice*. Springer, New York.

Hills, M., P. Klint, T. Van Der Storm, and J. Vinju (2011). "A case of visitor versus interpreter pattern". In: *Objects, Models, Components, Patterns*. Springer, pp. 228–243.

Hoch, F., M. Kerr, A. Griffith, et al. (2001). *Software as a service: Strategic backgrounder*. Tech. rep. Software & Information Industry Association (SIIA).

Homer, A., J. Sharp, L. Brader, N. Masashi, and S. Trent (2014). *Cloud Design Patterns: Prescriptive Architecture Guidance for Cloud Applications*. Microsoft Publishing, Redmont.

IBM (July 2011). *Best practices for cloud computing multi-tenancy*. http://www.ibm.com/developerworks/cloud/library/cl-multitenantcloud/.

ISO/IEC (2001). *ISO/IEC 9126-1: 2001*. International Organization for Standardization.

– (2011). *ISO/IEC 25010: 2011*. International Organization for Standardization.

Jansen, A., J. Van Der Ven, P. Avgeriou, and D. K. Hammer (2007). "Tool support for architectural decisions". In: *The Working IEEE/IFIP Conference on Software Architecture (WICSA)*. IEEE, pp. 4–14.

Jaring, M. and J. Bosch (2002). "Representing variability in software product lines: A case study". In: *Software Product Lines* 2379, pp. 219–245.

Kabbedijk, J., C. Bezemer, S. Jansen, and A. Zaidman (2014 (In Press)). "Defining Multi-Tenancy: A Structured Mapping Study on the Academic and the Industrial Perspective". In: *Journal of Systems and Software*.

Kabbedijk, J., R. van Donselaar, and S. Jansen (2014). "SPEM: A Software Pattern Evaluation Method". In: *Proceedings of the International Conferences on Pervasive Patterns and Applications (PATTERNS)*. IARIA, pp. 38–43.

Kabbedijk, J., M. Galster, and S. Jansen (2012). "Focus Group Report: Evaluating the Consequences of Applying Architectural Patterns". In: *Proceedings of the European conference on Pattern Languages of Programs (EuroPLoP)*.

Kabbedijk, J. and S. Jansen (2011). "Variability in Multi-Tenant Environments: Architectural Design Patterns from Industry". In: *Proceedings of the International Conference on Advances in conceptual modeling: recent developments and new directions (ER)*. Springer, pp. 151–160.

– (2012). "The Role of Variability Patterns in Multi-Tenant Business Software". In: *Proceedings of the WICSA/ECSA 2012 Companion Volume*. ACM, pp. 143–146.

Kabbedijk, J., M. Pors, S. Jansen, and S. Brinkkemper (2014). "Multi-Tenant Architecture Comparison". In: *Proceedings of the European conference on Software Architecture (ECSA)*. ACM, pp. 202–209.

Kabbedijk, J., T. Salfischberger, and S. Jansen (2013). "Comparing Two Architectural Patterns for Dynamically Adapting Functionality in Online Software Products - Best Paper Award". In: *Proceedings of the International Conferences on Pervasive Patterns and Applications (PATTERNS)*, pp. 20–25.

Kabbedijk, J., S. Jansen, and S. Brinkkemper (2012). "A Case Study of the Variability Consequences of the CQRS Pattern in Online Business Software". In: *Proceedings of the European Conference on Pattern Languages of Programs (EuroPLoP)*. ACM, 2:1–2:10.

Kabbedijk, J., S. Jansen, and T. Salfischberger (2014). "Runtime Variability in Online Software Products: A Comparison of Four Patterns". In: *International Journal On Advances in Software* 7.1 and 2, pp. 101–111.

Kan, S. H. (2002). *Metrics and models in software quality engineering*. Addison-Wesley, Boston.

Kazman, R., J. Asundi, and M. Klein (2001). "Quantifying the costs and benefits of architectural decisions". In: *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, pp. 297–306.

Kazman, R., M. Klein, and P. Clements (2000). *ATAM: Method for architecture evaluation*. Tech. rep. DTIC.

Kircher, M. and P. Jain (2004). *Pattern-Oriented Software Architecture, Volume 3: Patterns for Resource Management*. John Wiley & Sons, New York.

Kitchenham, B. (2004). *Procedures for performing systematic reviews*. Tech. rep. Keele University, UK.

Kitchenham, B. A., D. Budgen, and P. Brereton (2010). "The value of mapping studies: a participant-observer case study". In: *Proceedings of Evaluation and Assessment of Software Engineering (EASE)*, pp. 25–33.

Kitchenham, B. A. and S. Charters (2007). *Guidelines for performing systematic literature reviews in software engineering*. Tech. rep. Keele University, UK.

Kitchenham, B., O. Pearl Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman (2009). "Systematic literature reviews in software engineering–a systematic literature review". In: *Information and software technology* 51.1, pp. 7–15.

Kitchenham, B. and S. L. Pfleeger (1996). "Software quality: The elusive target". In: *IEEE software* 13.1, pp. 12–21.

Kitzinger, J. (1995). "Qualitative research. Introducing focus groups." In: *BMJ: British medical journal* 311.7000, pp. 299–302.

Koziolek, H. (2011). "The sposad architectural style for multi-tenant software applications". In: *Proceeding of the Working IEEE/IFIP Conference on Software Architecture (WICSA)*. IEEE, pp. 320–327.

Krasner, G. and S. Pope (1988). "A description of the model-view-controller user interface paradigm in the smalltalk-80 system". In: *Journal of Object Oriented Programming* 1.3, pp. 26–49.

Kwok, T., T. Nguyen, and L. Lam (2008). "A software as a service with multi-tenancy support for an electronic contract management application". In: *Proceedings of the International Conference on Services Computing (SCC)*. IEEE, pp. 179–186.

Lauder, A. and S. Kent (1998). "Precise visual specification of design patterns". In: *Proceeding of Object-Oriented Programming*. Springer, pp. 114–134.

Li, X., T. Liu, Y. Li, and Y. Chen (2008). "SPIN: Service performance isolation infrastructure in multi-tenancy environment". In: *Proceedings of the International Conference on Service-Oriented Computing (ICSOC)*. Springer, pp. 649–663.

Lin, H., K. Sun, S. Zhao, and Y. Han (2009). "Feedback-control-based performance regulation for multi-tenant applications". In: *Proceedings of the International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, pp. 134–141.

Ma, D. (2007). "The Business Model of Software-As-A-Service". In: *Proceedings of the International Conference on Services Computing (SCC)*. IEEE, pp. 701–702.

Manuel, P. and J. AlGhamdi (2003). "A data-centric design for n-tier architecture". In: *Information Sciences* 150.3, pp. 195–206.

Mapelsden, D., J. Hosking, and J. Grundy (2002). "Design pattern modelling and instantiation using DPML". In: *Proceedings of the International Conference on Tools Pacific: Objects for internet, mobile and embedded applications*. Australian Computer Society, pp. 3–11.

Maplesden, D., J. G. Hosking, and J. C. Grundy (2001). "A Visual Language for Design Pattern Modelling and Instantiation." In: *Proceedings of the Symposia on Human-Centric Computing Languages and Environments*. IEEE, pp. 338–339.

March, S. T. and G. F. Smith (1995). "Design and natural science research on information technology". In: *Decision support systems* 15.4, pp. 251–266.

Mell, P. and T. Grance (2011). *The NIST definition of cloud computing*. Tech. rep. NIST.

Merriam Webster Online (2014a). *Evaluate*. http://www.merriam-webster.com/dictionary/evaluate/.

– (2014b). *Validate*. http://www.merriam-webster.com/dictionary/validate/.

Messerschmitt, D. G. and C. Szyperski (2005). *Software ecosystem: understanding an indispensable technology and industry*. The MIT Press, Cambridge.

Meyer, B. (1988). *Object-oriented software construction*. Prentice Hall PTR, New York.

Michalik, B., P. Avgeriou, D. Tofan, M. Galster, and D. Weyns (2014). "Variability in Software Systems - A Systematic Literature Review". In: *IEEE Transactions on Software Engineering* 40.3, pp. 282–306.

Microsoft (June 2012). *Principles of Sharing in a Shared First World: Multi-Tenancy*. http://www.microsoft.com/government/en-us/federal/futurefed/pages/details.aspx?Principles-of-Sharing-in-a-Shared-First-World:-Multi-Tenancy&blogid=158.

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

Mietzner, R., F. Leymann, and M. P. Papazoglou (2008). "Defining composite configurable SaaS application packages using SCA, variability descriptors and multi-tenancy patterns". In: *Proceedings on the International Conference on Internet and Web Applications and Services (ICIW)*. IEEE, pp. 156–161.

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

Mitchell, M. and J. Jolley (2012). *Research design explained*. Cengage Learning, Boston.

Momm, C. and R. Krebs (2011). "A Qualitative Discussion of Different Approaches for Implementing Multi-Tenant SaaS Offerings." In: *Proceedings of the Software Engineering (SE) Workshop*.

Nambisan, S. (2001). "Why service businesses are not product businesses". In: *MIT Sloan Management Review* 42.4, pp. 72–80.

Natis, Y. V. (2008). "Reference Architecture for Multitenancy: Enterprise Computing in the Cloud". In: *Gartner* 3, pp. 4–8.

Nijhof, M. (2010). *Elegant Code >CQRS >Event Sourcing*. http://elegantcode.com/2010/02/05/cqrs-event-sourcing/.

Oracle (Oct. 2009). *Multi-Tenancy (Non Virtualized)*. http://www.oracle.com/technetwork/topics/cloud/blueprint-multi-tenant-novm-089433.html.

Osipov, C., G. Goldszmidt, M. Taylor, and I. Poddar (2009). *Develop and Deploy Multi-Tenant Web-delivered Solutions using IBM middleware: Part 2: Approaches for enabling multi-tenancy*. Tech. rep. IBM Corporation.

Peffers, K., T. Tuunanen, M. A. Rothenberger, and S. Chatterjee (2007). "A design science research methodology for information systems research". In: *Journal of management information systems* 24.3, pp. 45–77.

Perepletchikov, M., C. Ryan, K. Frampton, and Z. Tari (2007). "Coupling metrics for predicting maintainability in service-oriented designs". In: *Proceedings of the Australian Software Engineering Conference (ASWEC)*. IEEE, pp. 329–340.

Petersen, K., R. Feldt, S. Mujtaba, and M. Mattsson (2008). "Systematic mapping studies in software engineering". In: *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering*. ACM, pp. 71–80.

Petticrew, M. and H. Roberts (2009). *Systematic reviews in the social sciences: A practical guide*. Oxford: Blackwell Publisher, Hoboken, p. 336.

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

Pors, M., L. Blom, J. Kabbedijk, and S. Jansen (2013). *Sharing is Caring - A Decision Support Model for Multi-Tenant Architectures*. Tech. rep. UU-CS-2013-015. Department of Information and Computing Sciences, Utrecht University.

Pressman, R. S. (1994). *Software Engineering: a practitioner's approach*. McGraw-Hill, New York.

Reason, P. (1994). *Three approaches to participative inquiry*. Sage Publications, Thousand Oaks.

Rimal, B., E. Choi, and I. Lumb (2009). "A taxonomy and survey of cloud computing systems". In: *Proceedings of the International Joint Conference on INC, IMS and IDC (NCM)*. IEEE, pp. 44–51.

Rumbaugh, J., I. Jacobson, and G. Booch (2004). *The Unified Modeling Language Reference Manual*. Pearson Higher Education, London, UK.

Runeson, P. and M. Höst (2009). "Guidelines for conducting and reporting case study research in software engineering". In: *Empirical Software Engineering* 14.2, pp. 131–164.

S. Jansen G.J. Houben, S. B. (July 2010). "Customization Realization in Multi-tenant Web Applications: Case Studies from the Library Sector". In: *Proceedings of the 10th International Conference on Web Engineering (ICWE 2010)*. Vol. 6189. LNCS. Vienna, Austria: Springer, pp. 445–459.

Sääksjärvi, M., A. Lassila, and H. Nordström (2005). "Evaluating the software as a service business model: From CPU time-sharing to online innovation sharing". In: *Proceedings of the International Conference on e-Society*. IEEE, pp. 27–30.

Sandhu, R. S. and P. Samarati (1994). "Access control: principle and practice". In: *IEEE Communications Magazine* 32.9, pp. 40–48.

Schiller, O., B. Schiller, A. Brodt, and B. Mitschang (2011). "Native support of multi-tenancy in RDBMS for software as a service". In: *Proceedings of the International Conference on Extending Database Technology*. ACM, pp. 117–128.

Schmidt, D., M. Stal, H. Rohnert, and F. Buschmann (2000). *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*. John Wiley & Sons, New York.

Schmidt, D. C. (1995). "Using design patterns to develop reusable object-oriented communication software". In: *Communications of the ACM* 38.10, pp. 65–74.

Schmidt, D. C., M. Fayad, and R. E. Johnson (1996). "Software patterns". In: *Communications of the ACM* 39.10, pp. 37–39.

Shadish, W. R., T. D. Cook, and D. T. Campbell (2002). *Experimental and quasi-experimental designs for generalized causal inference*. Cengage learning, New York.

Shaw, M. and D. Garlan (1996). *Software architecture: perspectives on an emerging discipline*. Prentice Hall, Englewood Cliffs.

Smith, M. A. and R. L. Kumar (2004). "A theory of application service provider (ASP) use from a client perspective". In: *Information & management* 41.8, pp. 977–1002.

Stevens, S. S. (1946). *On the theory of scales of measurement*. Bobbs-Merrill, Indianapolis.

Strauch, S., V. Andrikopoulos, S. Gómez Sáez, and F. Leymann (2013). "ESBMT: A Multi-tenant Aware Enterprise Service Bus". In: *International Journal of Next-Generation Computing* 4.3, pp. 230–249.

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

Svahnberg, M., J. van Gurp, and J. Bosch (2005). "A taxonomy of variability realization techniques". In: *Software: Practice and Experience* 35.8, pp. 705–754.

Swanson, E. B. and P. Wang (2005). "Knowing why and how to innovate with packaged business software". In: *Journal of Information Technology* 20.1, pp. 20–31.

Tao, L. (2001). "Shifting paradigms with the application service provider model". In: *Computer* 34.10, pp. 32–39.

Taylor, R., N. Medvidovic, and E. Dashofy (2010). *Software Architecture: Foundations, Theory and Practice*. John Wiley & Sons, New York.

Thiel, S. and A. Hein (2002). "Modeling and using product line variability in automotive systems". In: *IEEE software* 19.4, pp. 66–72.

Torkel, O. (2010). *Queries & Aggregates & DDD*. http://www.codinginstinct.com/2011/04/queries-aggregates-ddd.html.

Tremblay, M. C., A. R. Hevner, and D. J. Berndt (2010). "The Use of Focus Groups in Design Science Research". In: *Design Research in Information Systems* 22, pp. 121–143.

Tsai, C.-H., Y. Ruan, S. Sahu, A. Shaikh, and K. G. Shin (2007). "Virtualization-based techniques for enabling multi-tenant management tools". In: *Lecture Notes in Computer Science*. Vol. 4785. Springer, pp. 171–182.

Tucker, A. B. (2004). *Computer Science: Handbook*. CRC press, Boca Raton.

Turner, M., D. Budgen, and P. Brereton (2003). "Turning software into a service". In: *Computer* 36.10, pp. 38–44.

Tyree, J. and A. Akerman (2005). "Architecture decisions: Demystifying architecture". In: *IEEE Software* 22.2, pp. 19–27.

Van der Aalst, W., A. ter Hofstede, and M. Weske (2003). "Business process management: A survey". In: *Business Process Management* 2678, pp. 1019–1019.

Van Gurp, J., J. Bosch, and M. Svahnberg (2001). "On the notion of variability in software product lines". In: *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA)*. IEEE, pp. 45–54.

Vaquero, L. M., L. Rodero-Merino, J. Caceres, and M. Lindner (2008). "A break in the clouds: towards a cloud definition". In: *ACM SIGCOMM Computer Communication Review* 39.1, pp. 50–55.

Wang, Z. H., C. J. Guo, B. Gao, W. Sun, Z. Zhang, and W. H. An (2008). "A study and performance evaluation of the multi-tenant data tier design patterns for service oriented computing". In: *Proceedings of the International Conference on e-Business Engineering (ICEBE)*. IEEE, pp. 94–101.

Weinreich, R., T. Ziebermayr, and D. Draheim (2007). "A versioning model for enterprise services". In: *Proceedings of the International Conference on Advanced Information Networking and Applications Workshops (AINAW)*. Vol. 2. IEEE, pp. 570–575.

Wellhausen, T. and A. Fießer (2011). "How to write a pattern". In: *Proceedings of the European conference on pattern languages of programs (EuroPLoP)*.

Wieringa, R., H. Heerkens, and B. Regnell (2009). "How to write and read a scientific evaluation paper". In: *Proceedings of the International Requirements Engineering Conference (RE)*. IEEE, pp. 361–364.

Wieringa, R. (2009). "Design science as nested problem solving". In: *Proceedings of the International Conference on Design Science Research in Information Systems and Technology*. ACM, p. 8.

Wilkes, M. V. (1975). *Time sharing computer systems*. Elsevier Science, New York.

Wu, W., L. Lan, and Y. Lee (2011). "Exploring decisive factors affecting an organization's SaaS adoption: A case study". In: *International Journal of Information Management* 31.6, pp. 556–563.

Xu, L. and S. Brinkkemper (2007). "Concepts of product software". In: *European Journal of Information Systems* 16.5, pp. 531–541.

Yin, R. (2009). *Case study research: Design and methods*. Sage Publications, New York.

Young, G. (2010). *CQRS and Event Sourcing*. http://codebetter.com/gregyoung/2010/02/13/cqrs-and-event-sourcing/.

Zhang, Q., L. Cheng, and R. Boutaba (2010). "Cloud computing: state-of-the-art and research challenges". In: *Journal of Internet Services and Applications* 1.1, pp. 7–18.

Appendix A

Pattern Catalogue

This catalogue gives an overview of all software patterns originating from this dissertation. Every pattern starts on a new page and discusses the *context*, *problem*, *solution* and *consequences* of the pattern. The patterns description style is based on the style presented in Chapter 3. The element 'Forces' is integrated with 'Solution' for the sake of brevity.

Customizable Data Views Pattern

Context

Design of a multi-tenant enterprise application.

Problem

It must be possible to provide tenants the ability to indicate and save his preferences on the representation of data shown. How can developers be enabled to give tenants a way to indicate their preferences on the representation of data within the software product?

Solution

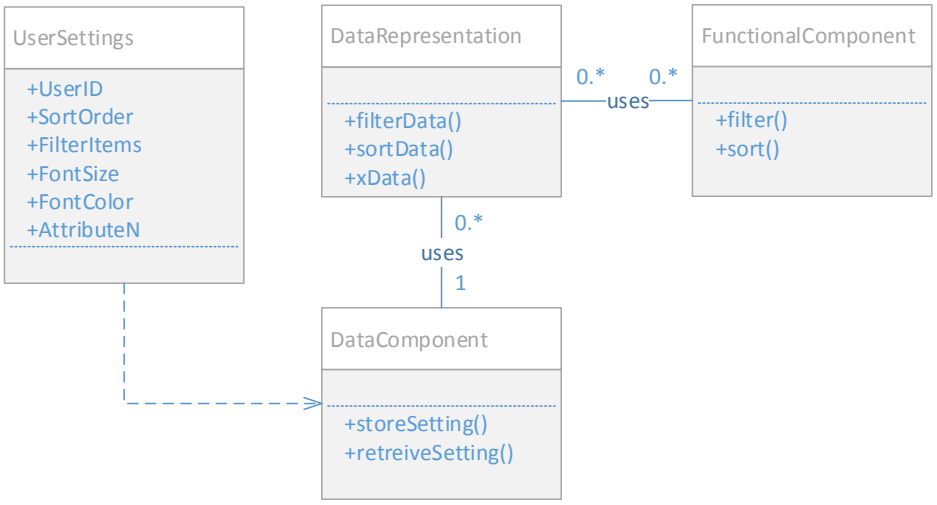


FIGURE A.1: Customizable Data Views Pattern

In this variability pattern, the representation of data is performed at client side. Tenants can for example choose how they want to sort or filter their data, while the data-queries do not have to be adapted. The only change needed to a software product is the introduction of tenant-specific representation settings. In this table, all preferred font colors, sizes and sort option can be stored in order to retrieve this information on other occasions to display the data again, according to the tenant's wishes. The *DataRepresentation* class can manipulate the appearance of all data by making use of a *FunctionalComponent* able of sorting, filtering, etcetera. All settings are later stored by a *DataComponent* in a specific *UserSettings* table.

Settings can later be retrieved by the same DataComponent, to be used again by the DataRepresentation class and FunctionalModule.

Consequences

By implementing this pattern, one extra table has to be implemented. Nothing changes in the way data selection queries have to be formatted. Representation of all data has to be formatted in a default way, except if a tenant changes this default way and stores his own preferences.

More details on this pattern can be found on page 64.

Module Dependent Menu Pattern

Context

Design of a multi-tenant enterprise application.

Problem

Tenants of a shared software system have specific requirements to a software product, they can all use different sets of functionality. Displaying all possible functionality in the menu would decrease the user experience of tenants, so menus have to display only the functionality that is relevant to the tenant. How can a custom menu to all tenants, only containing links to the functionality relevant to the tenant be provided?

Solution

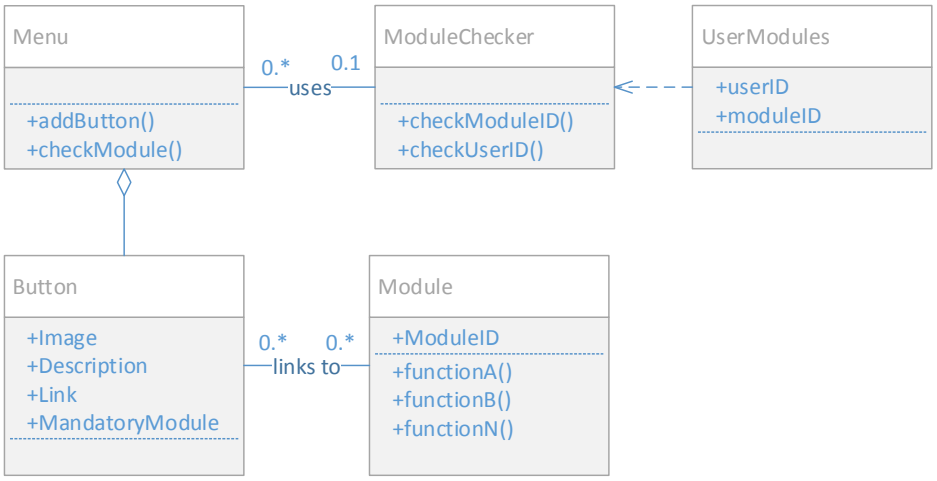


FIGURE A.2: Module Dependent Menu Pattern

The pattern creates a menu out of different buttons based on the modules associated to the tenant. Every time a tenant displays the menu, the menu is built dynamically based on the modules he has selected or bought. The *Menu* class aggregates and displays different *buttons*, containing a link a specific module and the prerequisite for displaying this link (*mandatoryModule*). The selection of buttons is done, based on the results of the *ModuleChecker*. This class checks whether an entry is available in the *UserModules* table, containing both the ID of the tenant

(user) and the mandatory module. If an entry is present, the *Menu* aggregates and displays the button corresponding to this module.

Consequences

To be able to use this pattern, an extra table containing user IDs and the modules available to this user has to be implemented. Also, the extra class *ModuleChecker* has to be implemented. All buttons do need a notion of a mandatory module that can be checked by the ModuleChecker to verify if a tenant wants or can have a link to the specific functionality.

More details on this pattern can be found on page 65.

Pre/Post Update Hooks Pattern

Context

Design of a multi-tenant enterprise application.

Problem

In business oriented software, workflows often differ per tenant. To let the software product fit the tenants business processes best, extra actions could be made available to tentants before or after an event is called. How can the possibility for tenants to have custom functionality just before or after an event be provided?

Solution

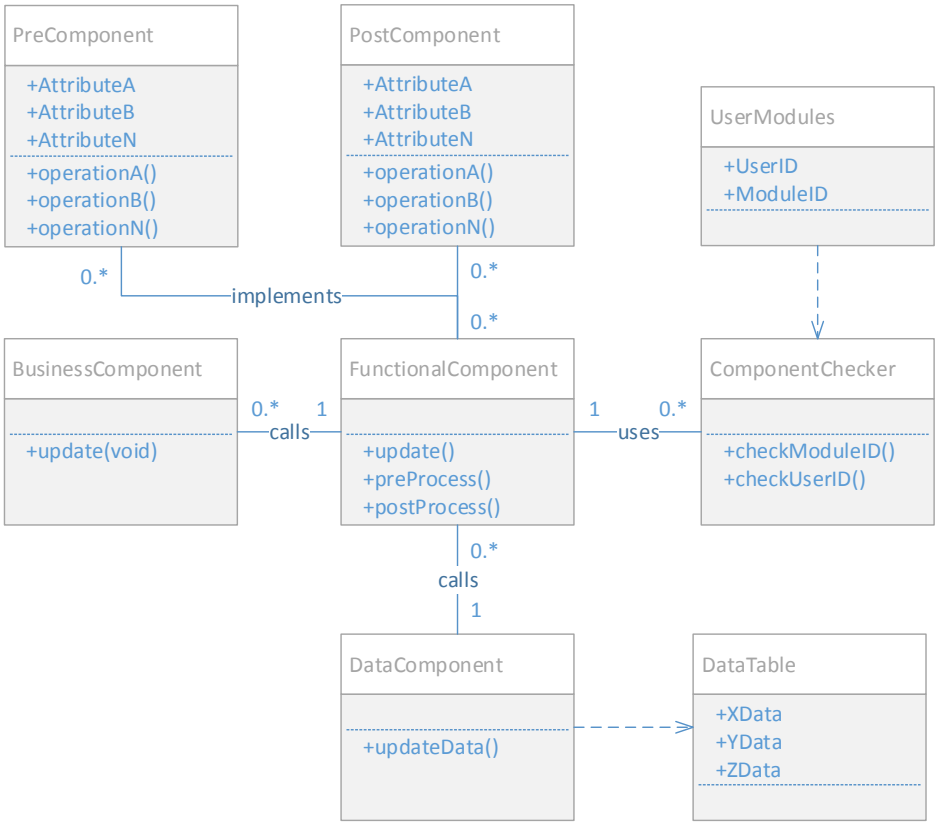


FIGURE A.3: Pre/Post Update Hooks Pattern

The pattern creates a menu out of different buttons based on the modules associated to the tenant. Every time a tenant displays the menu, the menu is built

dynamically based on the modules he has selected or bought. The *Menu* class aggregates and displays different *buttons*, containing a link a specific module and the prerequisite for displaying this link (*mandatoryModule*). The selection of buttons is done, based on the results of the *ModuleChecker*. This class checks whether an entry is available in the *UserModules* table, containing both the ID of the tenant (user) and the mandatory module. If an entry is present, the *Menu* aggregates and displays the button corresponding to this module.

Consequences

To be able to use this pattern, an extra table containing user IDs and the modules available to this user has to be implemented. Also, the extra class *ModuleChecker* has to be implemented. All buttons do need a notion of a mandatory module that can be checked by the ModuleChecker to verify if a tenant wants or can have a link to the specific functionality.

More details on this pattern can be found on page 67.

CQRS Pattern

Context

Design of a multi-tenant enterprise application.

Problem

In order to create a software product, capable of offering a certain level of variability, most current software products separate logic into different layers. Each tier within this architectural principle is responsible for a different part of the architecture. An often implemented solution to this multi-tier architecture is the three-tiered application in which there is a separate data, logic and presentation tier. Within this solution, the database in the data tier is often seen as one CRUD (i.e. Create, Read, Update and Delete data) data store in which all commands and queries are performed on the same database. This can lead to locking, performance and scalability problems, especially with larger commands or queries, since all things have to be taken care of sequentially. Distributing parts of the system in combination with selective locking of data provides a partial solution, but leads to a high probability of data inconsistency.

Solution

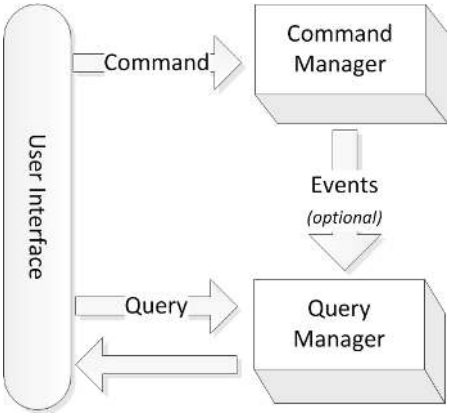


Figure A.4: CQRS Pattern

The CQRS pattern is prescribes the creation of two subsystems in a system design. From the user interfaces commands can be sent to the command manager or queries can be send to or received from the query manager. Commands are actions that will be performed on the data, while queries are requests for data to be shown. The

CQRS pattern itself does not prescribe anything about communication between the command manager and the query manager, but the following sub patterns are often used in combination with CQRS:

- Event Sourcing
- Event Store
- Aggregate Root
- Command Handler
- Query Model Builder
- Query Handler
- Snapshotting

A description of the patterns mentioned above can be found in the sections of the pattern catalogue.

Consequences

By implementing CQRS, highly scalable and variable multi-tenant enterprise products can be designed. Also, it enables for the use of the aforementioned sub patterns.

More details on this pattern can be found in Chapter 5.

Event Sourcing Pattern

Context

Design of a multi-tenant enterprise application, applying the CQRS Pattern.

Problem

There needs to be a way to communicate between the command manager and the query manager.

Solution

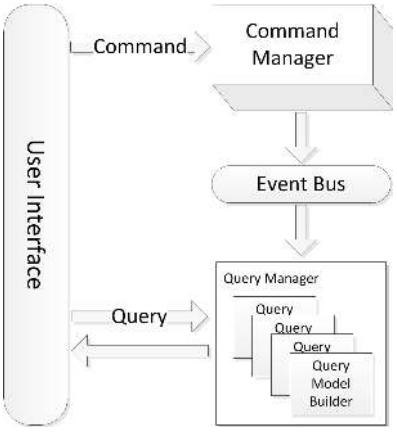


Figure A.5: Event Sourcing Pattern

Sourcing of the events created by the command manager, which can be sent to an event bus to which the QUERY MODEL BUILDERS in the query manager listen. The different query builders can be on the same system, but also on different physical or virtual machines. Query model builders can be on different geographical locations or even at clients. The most important aspect of the event sourcing pattern is the fact different events are broadcasted by the command manager to be processed by different components.

Consequences

The system becomes scalable and all sub parts are specially geared towards the task they have to do (ie. read or write).

More details on this pattern can be found on page 77.

Event Store Pattern

Context

Design of a multi-tenant enterprise application, applying the CQRS Pattern.

Problem

Storage by the query manager could be anything, from stored in cache, to stored at the client, or in some database. Because of the uncertainty in storing method, you can not rely on the availability and recovery of data if the system crashes.

Solution

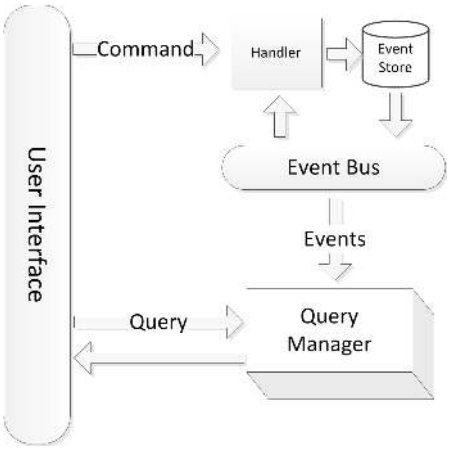


Figure A.6: Event Sourcing Pattern

In the event store all events can be stored sequentially, so the all data can be reconstructed based on the events in case of a system crash. From the *User Interface* commands are sent to a *handler*, who sends it to the event store as an event.

Consequences

Events are now stored in a central location and can be accessed in a reliable way, for the sake of data recovery.

More details on this pattern can be found on page 77.

Aggregate Root Pattern

Context

Design of a multi-tenant enterprise application, applying the CQRS Pattern.

Problem

Because data in the CQRS pattern is created by different query model builders of which you do not necessarily know what or where they are, and because of the asynchronous way these listeners work you can not say anything about the correctness of data at the time of querying. As an example, think about a large web shop selling laptops. Whenever someone wants to order a laptop, the system needs to know whether the inventory is sufficient to approve the order. In other words, the system needs to be sure there is at least one laptop available before the order can be processed. In the core CQRS pattern, there is no way to know for sure the laptop is in stock, because all events are processed asynchronous. The only way to know for sure the laptop is in stock, is to store the number of laptops available together with the laptop itself and also process this as one. If not, it is possible that the system checks whether a laptop is in stock, sees one laptop in stock, starts processing the order and ends up with an erroneous order since the laptop is sold just before through another process.

Solution

The concept of storing and processing all properties and entities that are dependent on each other together is know as aggregation. The main entity is called the entity root. An order, for example should always be processed together with its order lines, since the lines make no sense without the order. In the previously mentioned example, the order and order lines are an *aggregate* and the order is the *aggregate root*, since deleting the root would indicate deleting the other entities as well.

Consequences

Related properties and entities are processed and stored together.

More details on this pattern can be found on page 78.

Command Handler Pattern

Context

Design of a multi-tenant enterprise application, applying the CQRS Pattern.

Problem

Commands coming in from the user interface have to be passed through to something that will perform the action dictated by the command. These actions can be adequately performed by Aggregate roots, but the commands coming from the command bus have to be interpreted and translated somehow before they can be processed.

Solution

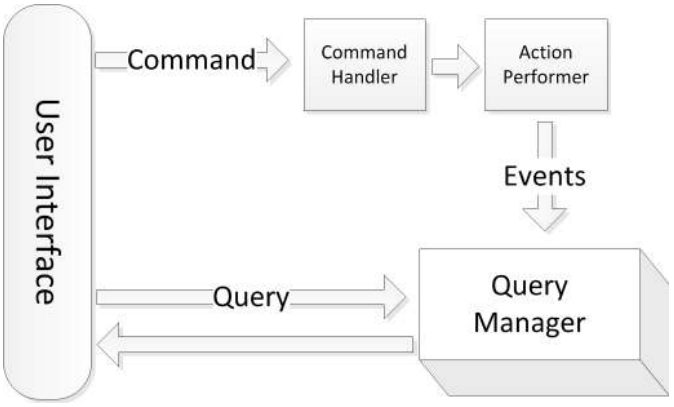


Figure A.7: Command Handler Pattern

A command handler is capable of catching one or more commands and passing it through to an object capable of performing the command. The action performer makes sure an action is actually performed, by, for example, delaying the sourcing of events until an aggregate root is completely finished.

Consequences

Commands are correctly and timely processed.

More details on this pattern can be found on page 79.

Query Manager Pattern

Context

Design of a multi-tenant enterprise application, applying the CQRS Pattern.

Problem

Data queries by tenants are diverse and need to be translated to an appropriate view. In order to represent the right data in the appropriate form, the needed view is dependent on the domain of the query. The domain knowledge needs to be translated to an automatically usable model.

Solution

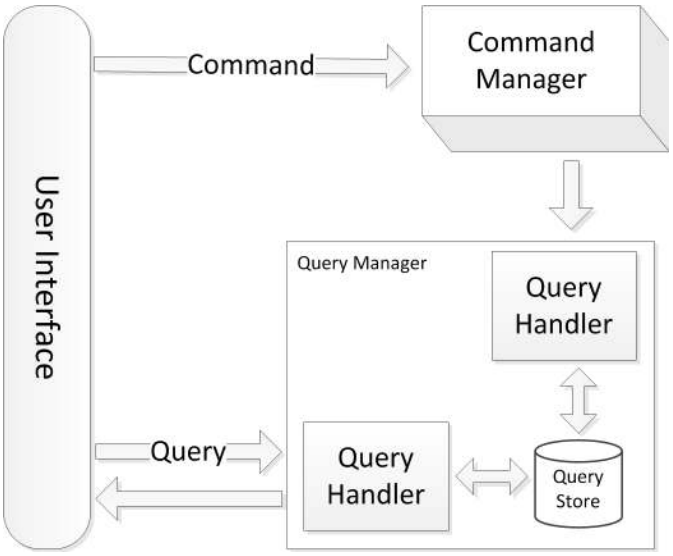


Figure A.8: QMB Manager Pattern

The query handler implements a component able of receiving all queries and checking the query store for views created by the Query Model Builder. Query Model Builders (QMBs) can be everywhere, from the clients cache, to on all kind of different physical servers. The QMBs listen to events coming in through the event bus, and create a view of the data needed by the query manager. This view totally depends on the domain the QMB is in and the goal the data has. A QMB in a system responsible for generating inventories, for example, will build entirely different query models than a QMB in a system responsible for displaying the contact details of one person. The concept of a query store is introduced to

store queries build by the QMB. This store is not obligatory, but can improve the response time of the system.

Consequences

Queries are now translated to views, usable for representation to tenants through the user interface.

More details on this pattern can be found in Section 5.5.5 and 5.5.6 on page 80.

Snapshotting Pattern

Context

Design of a multi-tenant enterprise application, applying the CQRS Pattern.

Problem

It is common practice in the CQRS pattern to only store changes (events) and no states. This is because states can always be determined based on all the changes happened in the system so far. Rerunning al events will bring the system back in its last state after a possible system crash. States only occur in aggregate roots, but recovering the state of an aggregate root after a system crash can be quite intensive, since aggregate roots often stay active in the system for a long time.

Solution

In the snapshotting pattern, the state of the aggregate root is stored together with the events every n^{th} event. The exact value of n depends on the processing load storing and monitoring the state of the aggregate root gives. When the system crashes, the latest stored state is recovered and only the events happened after this state storage have to be rerun. The snapshotting pattern is often used in combination with the memento pattern (Gamma, Helm, Johnson, and Vlissides, 1995) that provides the ability to restore objects to their previous state.

Consequences

System recovery is faster and more reliable.

More details on this pattern can be found on page 81.

Dedicated Application and Database Server Pattern

Context

Design of the architecture of a multi-tenant enterprise application.

Problem

In order to implement a multi-tenant architecture, resources have to be shared on different levels of the computing stack. It is not clear what can be shared and what the effects are of sharing specific levels of the computing stack.

Solution

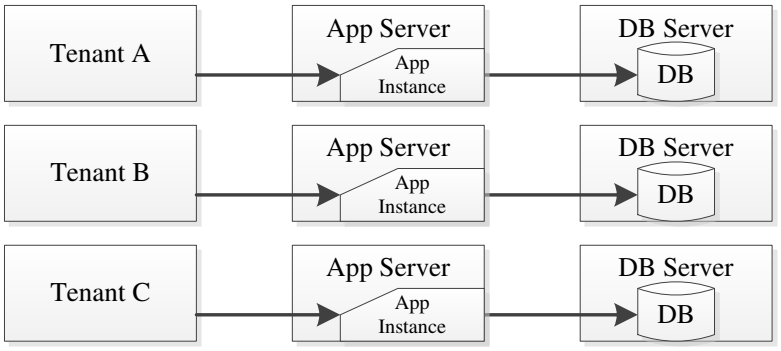


FIGURE A.9: Dedicated Application and Database Server Pattern

The tenants share no resources at all. For each tenant, a dedicated application server and a dedicated database server is run.

Consequences

This pattern has the following consequences, with **1** being a high **negative** influence and **5** a high **positive** influence:

	Time Behavior	Resource Utilization	Throughput	Number of Tenants	Number of End-Users	Availability	Recoverability	Confidentiality	Integrity	Authenticity	Maintainability	Portability	Deployment Time	Variability	Diverse SLA	Software Complexity	Monitoring
Rating	5.0	2.5	4.5	1.0	2.5	4.0	5.0	5.0	4.5	4.5	1.5	5.0	1.5	5.0	4.5	5.0	1.0

TABLE A.1: MTA $\langle AD, DD \rangle$ Pattern Consequences (In color)

Applying this pattern has a positive effect on the variability and recoverability of a software product, but harms the maintainability and the number of tenants that can be served.

More details on this pattern can be found in Chapter 6.

Shared Application Server / Dedicated Database Server Pattern

Context

Design of the architecture of a multi-tenant enterprise application.

Problem

In order to implement a multi-tenant architecture, resources have to be shared on different levels of the computing stack. It is not clear what can be shared and what the effects are of sharing specific levels of the computing stack.

Solution

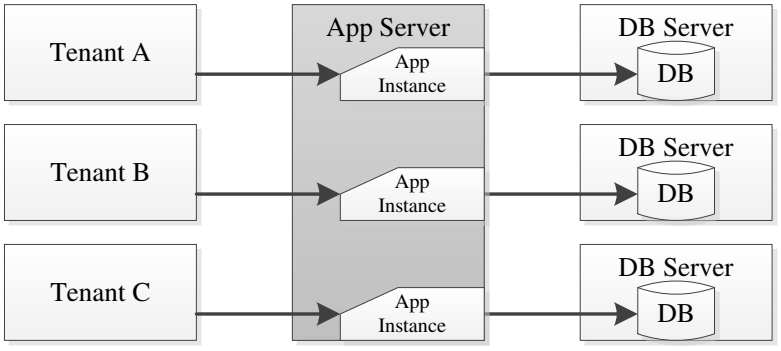


FIGURE A.10: Shared Application Server / Dedicated Database Server Pattern

Tenants only share an application server. A dedicated application instance and database server is running for each tenant.

Consequences

This pattern has the following consequences, with **1** being a high **negative** influence and **5** a high **positive** influence:

	Time Behavior	Resource Utilization	Throughput	Number of Tenants	Number of End-Users	Availability	Recoverability	Confidentiality	Integrity	Authenticity	Maintainability	Portability	Deployment Time	Variability	Diverse SLA	Software Complexity	Monitoring
Rating	4.0	2.5	3.0	3.0	3.5	3.0	4.5	4.5	4.0	3.5	2.5	5.0	3.0	4.0	4.0	4.5	2.0

TABLE A.2: MTA $\langle AS, DD \rangle$ Pattern Consequences (In color)

Applying this pattern has a positive effect on the portability and recoverability of a software product, but harms the monitoring and resource utilization.

More details on this pattern can be found in Chapter 6.

Shared Instance / Dedicated Database Server Pattern

Context

Design of the architecture of a multi-tenant enterprise application.

Problem

In order to implement a multi-tenant architecture, resources have to be shared on different levels of the computing stack. It is not clear what can be shared and what the effects are of sharing specific levels of the computing stack.

Solution

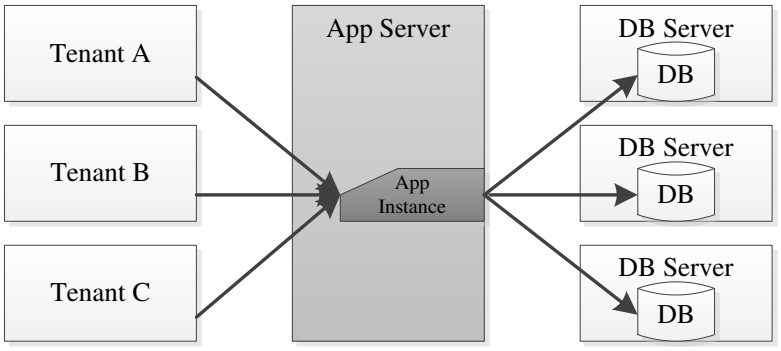


FIGURE A.11: Shared Instance / Dedicated Database Server Pattern

Tenants share an application server and instance. A dedicated database server is running for each tenant.

Consequences

This pattern has the following consequences, with **1** being a high **negative** influence and **5** a high **positive** influence:

	Time Behavior	Resource Utilization	Throughput	Number of Tenants	Number of End-Users	Availability	Recoverability	Confidentiality	Integrity	Authenticity	Maintainability	Portability	Deployment Time	Variability	Diverse SLA	Software Complexity	Monitoring
Rating	4.0	3.0	3.0	3.0	3.0	3.0	4.5	4.0	3.0	3.0	3.0	4.5	3.0	2.5	3.0	4.0	3.0

TABLE A.3: MTA $\langle AI, DD \rangle$ Pattern Consequences (In color)

Applying this pattern has a positive effect on the portability and recoverability of a software product, but harms the variability.

More details on this pattern can be found in Chapter 6.

Dedicated Application Server / Shared Database Server Pattern

Context

Design of the architecture of a multi-tenant enterprise application.

Problem

In order to implement a multi-tenant architecture, resources have to be shared on different levels of the computing stack. It is not clear what can be shared and what the effects are of sharing specific levels of the computing stack.

Solution

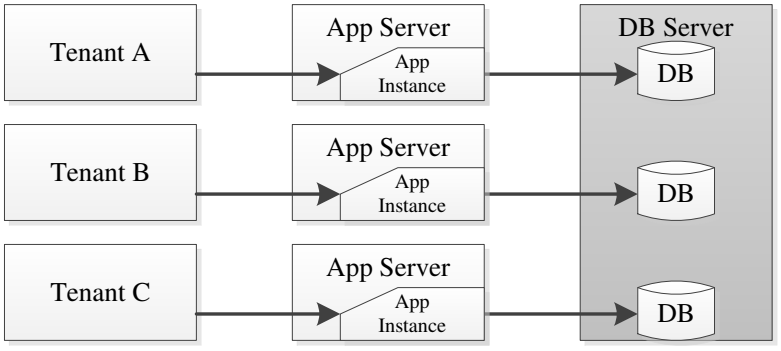


FIGURE A.12: Dedicated Application Server / Shared Database Server Pattern

Tenants share a database server, but each have their own database. A dedicated application server is running for each tenant.

Consequences

This pattern has the following consequences, with **1** being a high **negative** influence and **5** a high **positive** influence:

	Time Behavior	Resource Utilization	Throughput	Number of Tenants	Number of End-Users	Availability	Recoverability	Confidentiality	Integrity	Authenticity	Maintainability	Portability	Deployment Time	Variability	Diverse SLA	Software Complexity	Monitoring
Rating	4.0	2.5	4.0	3.0	3.0	3.0	4.0	4.0	4.0	3.5	2.5	4.5	2.5	5.0	4.0	4.5	2.5

TABLE A.4: MTA $\langle AD, DS \rangle$ Pattern Consequences (In color)

Applying this pattern has a positive effect on the variability and portability of a software product, but harms the deployment time and resource utilization.

More details on this pattern can be found in Chapter 6.

Shared Application and Database Server Pattern

Context

Design of the architecture of a multi-tenant enterprise application.

Problem

In order to implement a multi-tenant architecture, resources have to be shared on different levels of the computing stack. It is not clear what can be shared and what the effects are of sharing specific levels of the computing stack.

Solution

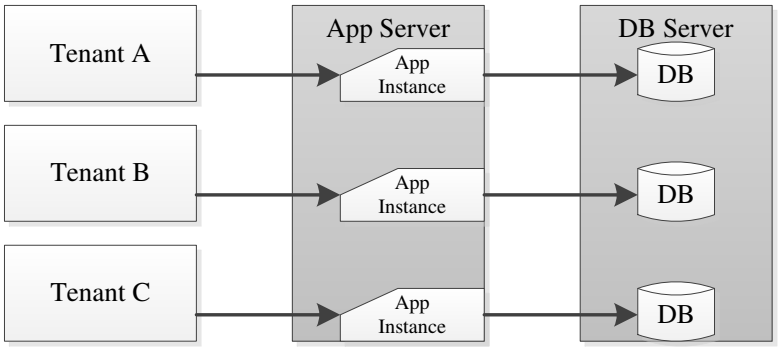


FIGURE A.13: Shared Application and Database Server Pattern

Tenants share a database and application server, but all tenants have their own database and application instance.

Consequences

This pattern has the following consequences, with **1** being a high **negative** influence and **5** a high **positive** influence:

	Time Behavior	Resource Utilization	Throughput	Number of Tenants	Number of End-Users	Availability	Recoverability	Confidentiality	Integrity	Authenticity	Maintainability	Portability	Deployment Time	Variability	Diverse SLA	Software Complexity	Monitoring
Rating	3.0	3.0	3.0	3.5	3.5	3.0	4.0	4.0	3.5	3.0	3.0	4.5	3.5	4.0	3.5	4.5	3.0

TABLE A.5: MTA $\langle AS, DS \rangle$ Pattern Consequences (In color)

Applying this pattern has a positive effect on the portability and complexity of a software product, but has a neutral effect on many other quality attributes.

More details on this pattern can be found in Chapter 6.

Shared Instance and Database Server Pattern

Context

Design of the architecture of a multi-tenant enterprise application.

Problem

In order to implement a multi-tenant architecture, resources have to be shared on different levels of the computing stack. It is not clear what can be shared and what the effects are of sharing specific levels of the computing stack.

Solution

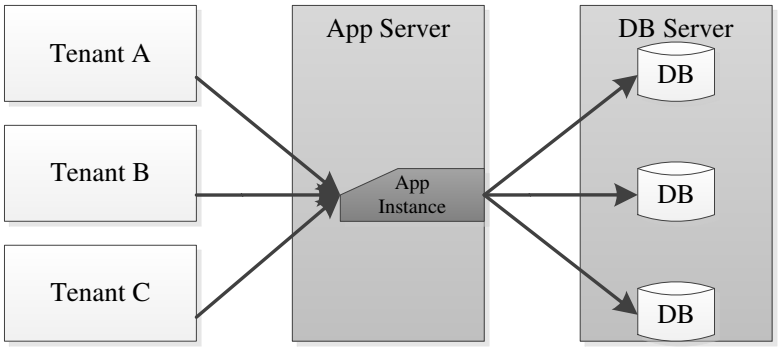


FIGURE A.14: Shared Instance and Database Server Pattern

Tenants share an application server, instance and database server. A dedicated database is running for each tenant.

Consequences

This pattern has the following consequences, with **1** being a high **negative** influence and **5** a high **positive** influence:

	Time Behavior	Resource Utilization	Throughput	Number of Tenants	Number of End-Users	Availability	Recoverability	Confidentiality	Integrity	Authenticity	Maintainability	Portability	Deployment Time	Variability	Diverse SLA	Software Complexity	Monitoring
Rating	3.0	3.0	3.0	4.0	3.5	3.0	4.0	4.0	3.0	3.0	3.5	4.5	4.0	2.0	2.5	3.5	3.0

TABLE A.6: MTA $\langle AI, DS \rangle$ Pattern Consequences (In color)

Applying this pattern has a positive effect on the portability of a software product, but harms the variability and diversification possibilities of the service level agreements.

More details on this pattern can be found in Chapter 6.

Dedicated Application Instance / Shared Database Pattern

Context

Design of the architecture of a multi-tenant enterprise application.

Problem

In order to implement a multi-tenant architecture, resources have to be shared on different levels of the computing stack. It is not clear what can be shared and what the effects are of sharing specific levels of the computing stack.

Solution

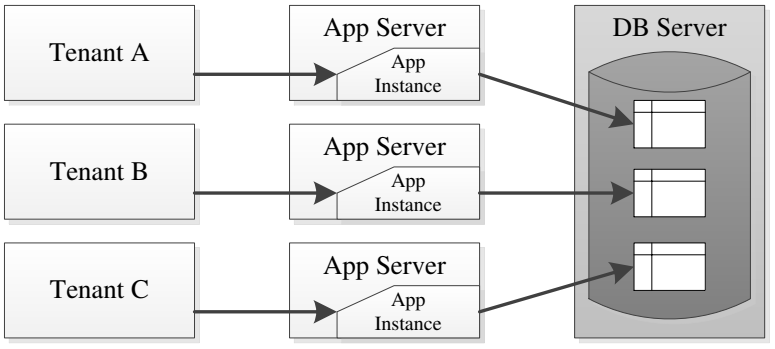


FIGURE A.15: Dedicated Application Instance / Shared Database Pattern

Tenants share a database and database server. A dedicated application server is running for each tenant.

Consequences

This pattern has the following consequences, with **1** being a high **negative** influence and **5** a high **positive** influence:

	Time Behavior	Resource Utilization	Throughput	Number of Tenants	Number of End-Users	Availability	Recoverability	Confidentiality	Integrity	Authenticity	Maintainability	Portability	Deployment Time	Variability	Diverse SLA	Software Complexity	Monitoring
Rating	4.0	3.0	3.5	3.0	3.5	3.0	3.0	3.5	3.5	4.0	2.5	4.0	3.0	4.5	4.0	4.0	3.0

TABLE A.7: MTA $\langle AD, DB \rangle$ Pattern Consequences (In color)

Applying this pattern has a positive effect on the variability of a software product, but harms the maintainability.

More details on this pattern can be found in Chapter 6.

Shared Application Server and Database Pattern

Context

Design of the architecture of a multi-tenant enterprise application.

Problem

In order to implement a multi-tenant architecture, resources have to be shared on different levels of the computing stack. It is not clear what can be shared and what the effects are of sharing specific levels of the computing stack.

Solution

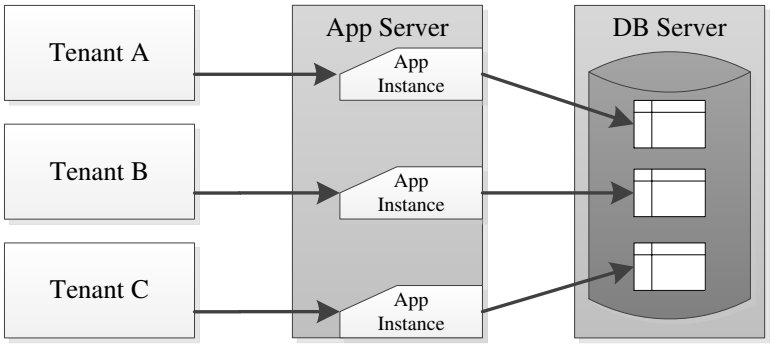


FIGURE A.16: Shared Application Server and Database Pattern

Tenants share a database, database server and application server. A dedicated application instance is running for each tenant.

Consequences

This pattern has the following consequences, with **1** being a high **negative** influence and **5** a high **positive** influence:

	Time Behavior	Resource Utilization	Throughput	Number of Tenants	Number of End-Users	Availability	Recoverability	Confidentiality	Integrity	Authenticity	Maintainability	Portability	Deployment Time	Variability	Diverse SLA	Software Complexity	Monitoring
Rating	3.0	3.0	3.5	4.0	3.5	3.0	3.0	3.0	3.0	3.5	4.0	4.0	4.0	3.5	3.0	4.0	4.0

TABLE A.8: MTA $\langle AS, DB \rangle$ Pattern Consequences (In color)

Applying this pattern has a slight positive effect on the maintainability and deployment time of a software product, but has a neutral effect on many other quality attributes.

More details on this pattern can be found in Chapter 6.

Shared Instance and Database Pattern

Context

Design of the architecture of a multi-tenant enterprise application.

Problem

In order to implement a multi-tenant architecture, resources have to be shared on different levels of the computing stack. It is not clear what can be shared and what the effects are of sharing specific levels of the computing stack.

Solution

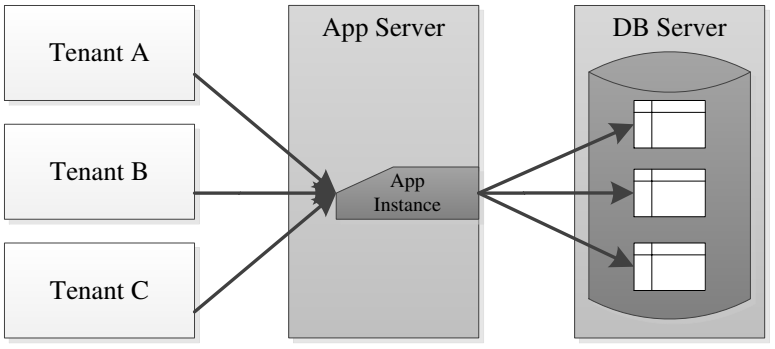


FIGURE A.17: Shared Instance and Database Pattern

Tenants share a database and application instance. Tenant data is stored in separate tables.

Consequences

This pattern has the following consequences, with **1** being a high **negative** influence and **5** a high **positive** influence:

	Time Behavior	Resource Utilization	Throughput	Number of Tenants	Number of End-Users	Availability	Recoverability	Confidentiality	Integrity	Authenticity	Maintainability	Portability	Deployment Time	Variability	Diverse SLA	Software Complexity	Monitoring
Rating	3.0	4.0	3.0	4.0	4.0	3.5	3.0	3.0	3.0	3.0	4.5	4.0	4.0	2.0	3.0	3.0	4.0

TABLE A.9: MTA $\langle AI, DB \rangle$ Pattern Consequences (In color)

Applying this pattern has a slight positive effect on the maintainability and resource utilization of a software product, but harms the variability.

More details on this pattern can be found in Chapter 6.

Dedicated Application Server / Shared Schema Pattern

Context

Design of the architecture of a multi-tenant enterprise application.

Problem

In order to implement a multi-tenant architecture, resources have to be shared on different levels of the computing stack. It is not clear what can be shared and what the effects are of sharing specific levels of the computing stack.

Solution

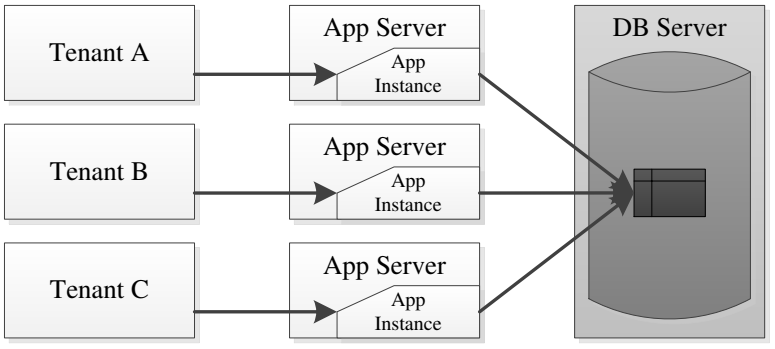


FIGURE A.18: Dedicated Application Server / Shared Schema Pattern

Tenants share a database, database schema and database server. A dedicated application server is running for each tenant.

Consequences

This pattern has the following consequences, with **1** being a high **negative** influence and **5** a high **positive** influence:

	Time Behavior	Resource Utilization	Throughput	Number of Tenants	Number of End-Users	Availability	Recoverability	Confidentiality	Integrity	Authenticity	Maintainability	Portability	Deployment Time	Variability	Diverse SLA	Software Complexity	Monitoring
Rating	3.5	3.0	3.0	4.0	3.5	3.0	2.0	2.0	3.0	3.0	3.0	3.0	3.0	2.5	3.0	2.5	3.5

TABLE A.10: MTA $\langle AD, DC \rangle$ Pattern Consequences (In color)

Applying this pattern harms the recoverability and confidentiality of a software product, but has a neutral effect on many other quality attributes.

More details on this pattern can be found in Chapter 6.

Shared Application Server and Database Schema Pattern

Context

Design of the architecture of a multi-tenant enterprise application.

Problem

In order to implement a multi-tenant architecture, resources have to be shared on different levels of the computing stack. It is not clear what can be shared and what the effects are of sharing specific levels of the computing stack.

Solution

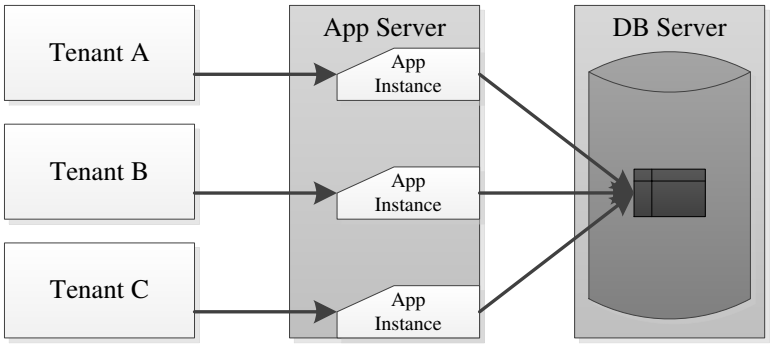


FIGURE A.19: Shared Application Server and Database Schema Pattern

Tenants share a database, database schema and database server. A dedicated application instance is running for each tenant.

Consequences

This pattern has the following consequences, with **1** being a high **negative** influence and **5** a high **positive** influence:

	Time Behavior	Resource Utilization	Throughput	Number of Tenants	Number of End-Users	Availability	Recoverability	Confidentiality	Integrity	Authenticity	Maintainability	Portability	Deployment Time	Variability	Diverse SLA	Software Complexity	Monitoring
Rating	3.0	3.0	3.0	4.0	4.0	3.0	2.0	2.0	2.5	3.0	4.0	3.0	4.0	2.0	2.5	2.5	4.0

TABLE A.11: MTA $\langle AS, DC \rangle$ Pattern Consequences (In color)

Applying this pattern has a positive effect on the maintainability and deployment time of a software product, but harms the variability and recoverability.

More details on this pattern can be found in Chapter 6.

Shared Instance and Database Schema Pattern

Context

Design of the architecture of a multi-tenant enterprise application.

Problem

In order to implement a multi-tenant architecture, resources have to be shared on different levels of the computing stack. It is not clear what can be shared and what the effects are of sharing specific levels of the computing stack.

Solution

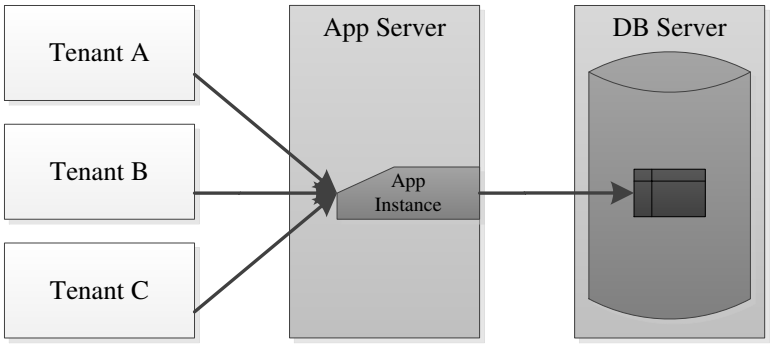


FIGURE A.20: Shared Instance and Database Schema Pattern

Tenants share an application instance and a database to schema level. No dedicated services are run.

Consequences

This pattern has the following consequences, with **1** being a high **negative** influence and **5** a high **positive** influence:

	Time Behavior	Resource Utilization	Throughput	Number of Tenants	Number of End-Users	Availability	Recoverability	Confidentiality	Integrity	Authenticity	Maintainability	Portability	Deployment Time	Variability	Diverse SLA	Software Complexity	Monitoring
Rating	2.5	4.5	3.0	5.0	4.5	3.0	2.0	2.5	2.0	2.5	5.0	2.5	5.0	1.0	2.0	2.0	5.0

TABLE A.12: MTA $\langle AI, DC \rangle$ Pattern Consequences (In color)

Applying this pattern has a positive effect on the maintainability and monitoring of a software product, but harms the variability and software complexity.

More details on this pattern can be found in Chapter 6.

Component Interceptor Pattern

Context

Design of a multi-tenant enterprise application.

Problem

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

Solution

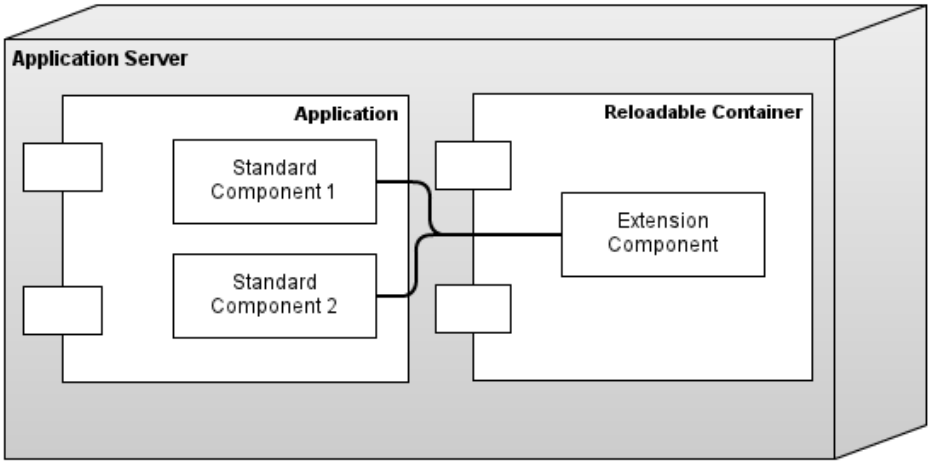


FIGURE A.21: Component Interceptor Pattern: System Model

Interceptors are tightly integrated with the application, because they run in-line with normal application code. Before the *StandardComponent* is called the interceptors are allowed to inspect and possibly modify the set of arguments and data passed to the standard component. To do this the interceptor has to be able to access all arguments, modify them or pass them along in the original form. Running interceptors outside of the application requires marshalling of the arguments and data to a format suitable for transport, then unmarshalling by the interceptor component and again marshalling the possibly modified arguments to be passed

on to the standard component that was being intercepted. This is impractical and involves a performance penalty.

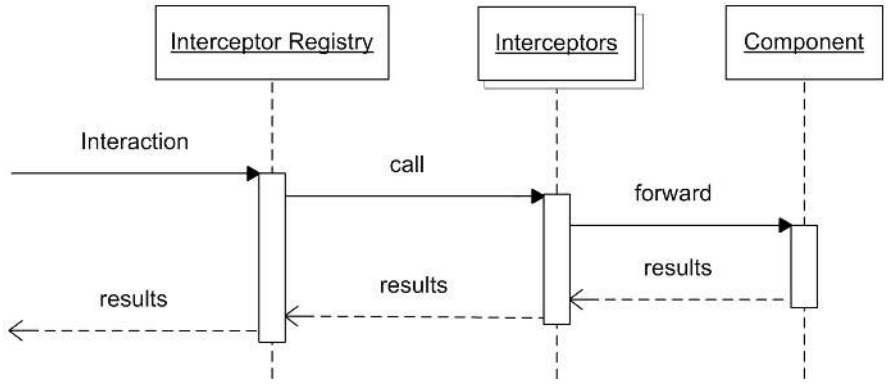


FIGURE A.22: Component Interceptor Pattern: Sequence Diagram

Interaction with standard components that can be extended goes through the interceptor registry. This registry is needed to keep track of all interceptors that are interested in each interaction. Without the registry the calling code would have to be aware of all possible interceptors. As depicted, multiple interceptors can be active per component. It is up to the interceptor registry to determine the order in which interceptors will be called. 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. As a result of these possibilities the interceptors must be invoked in-line with the standard component, the application cannot continue until all interceptors have finished executing.

Consequences

Security	- Extension components execute within application scope
Performance	+ Direct execution of extension components
Scalability	- No independent scaling of extension components - Does not scale to high number of extension components
Maintainability	- Tight coupling of extension components
Implementation Effort	+ Direct communication with standard components + Access to all data by design

TABLE A.13: Consequences of applying the Component Interceptor Pattern

More details on this pattern can be found on page 115.

Event Distribution Pattern

Context

Design of a multi-tenant enterprise application.

Problem

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

Solution

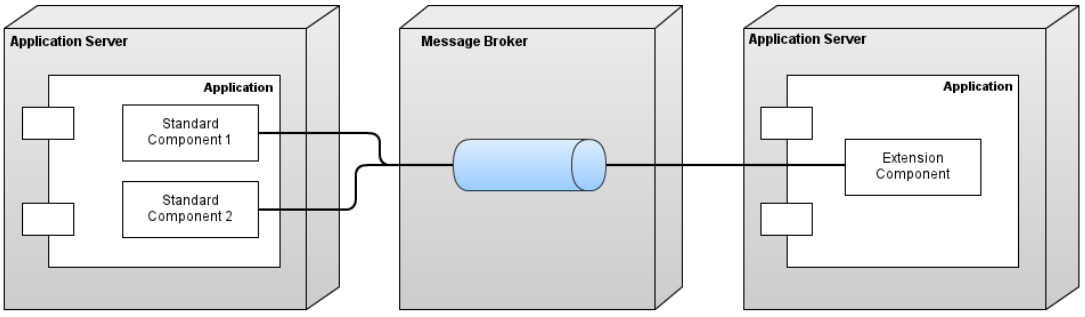


FIGURE A.23: Event Distribution Pattern: System Model

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

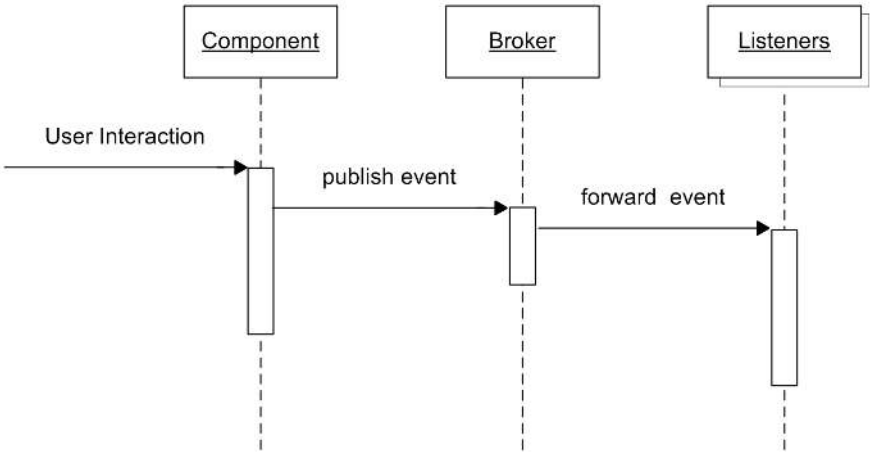


FIGURE A.24: Event Distribution Pattern: Sequence Diagram

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.

Consequences

Security	+ Isolation of extension components and full traceability
Performance	- Network overhead for calling extension components - The broker system requires extra resources
Scalability	+ Independent scaling of extension components + Extension components cannot delay standard components - Requires scalable message-broker system
Maintainability	+ Loose coupling of extension components
Implementation Effort	- Requires the setup of a message broker system - Requires a separate mechanism to communicate

TABLE A.14: Consequences of applying the Event Distribution Pattern

More details on this pattern can be found on page 116.

Datasource Router Pattern

Context

Design of a multi-tenant enterprise application.

Problem

Across markets and jurisdictions differences exist in regulations and standards which require the storage and reporting of different data for each organisation. Organisations will thus set varying requirements to store data specific to their needs. A software product that provides enough variability on the data model to meet organisation specific requirements will decrease cost and attract clients that cannot currently be serviced by software products unable to meet their specific requirements. Extension of the data model by creating additional fields to store data that are specific to an organisation or their working processes is a common requirement.

Solution

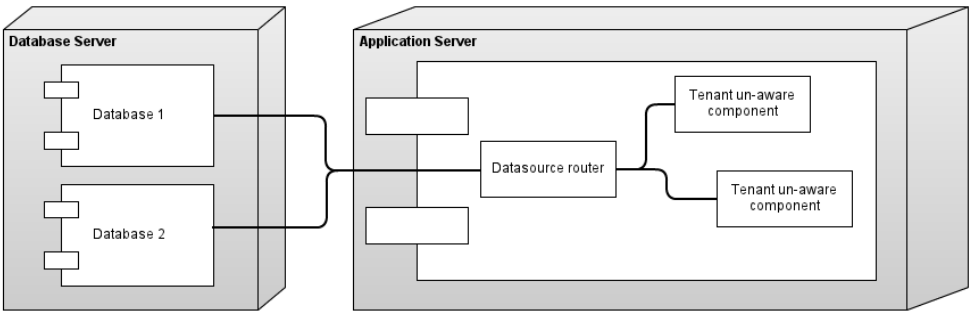


FIGURE A.25: Datasource Router Pattern: System Model

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

The interaction between tenant-unaware components and the database goes through the *Datasource Router*. First the user interacts with a component, this component requires access to data which is done through the *Datasource Router*. The

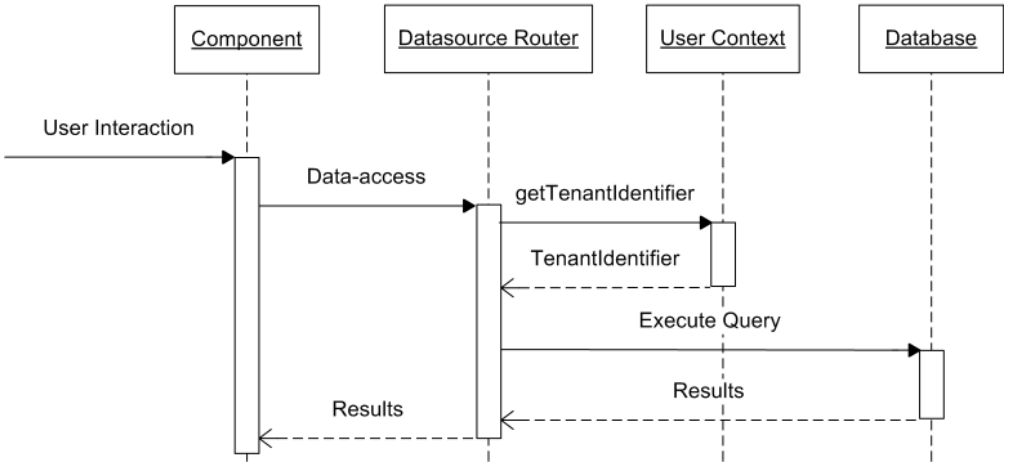


FIGURE A.26: Datasource Router Pattern: Sequence Diagram

Datasource Router is then responsible for determining which tenant the current user belongs to, this responsibility is delegated to the *User Context*. It is implementation dependant how this *User Context* is implemented, the only requirement is that it is able to tell the *Datasource Router* which tenant is to be used in the context of the current request. After determining which tenant is active the *Datasource Router* executes the query on the right database (selected based on the active tenant), the results are then returned to the component which originally needed access to the data. The component is isolated from these choices and the possible complexity involved in selecting the right datasource to use for the current user.

Consequences

Security	+ Natural separation of datasets + Single point of selecting correct datasource - More datasources to secure and maintain
Performance	+ Correct data-types allow for optimizations - Resource partitioning across separate schemas
Scalability	+ Natural scalability due to separate schemas + No need for scalability support in database
Maintainability	- Large number of possible database schemas must be tested - Problem solving requires schema variants to be included
Implementation Effort	+ Central component to handle all data-access - Custom properties must be handled in all components

TABLE A.15: Consequences of applying the Datasource Router Pattern

More details on this pattern can be found on page 124.

Custom Property Object Pattern

Context

Design of a multi-tenant enterprise application.

Problem

Across markets and jurisdictions differences exist in regulations and standards which require the storage and reporting of different data for each organisation. Organisations will thus set varying requirements to store data specific to their needs. A software product that provides enough variability on the data model to meet organisation specific requirements will decrease cost and attract clients that cannot currently be serviced by software products unable to meet their specific requirements. Extension of the data model by creating additional fields to store data that are specific to an organisation or their working processes is a common requirement.

Solution

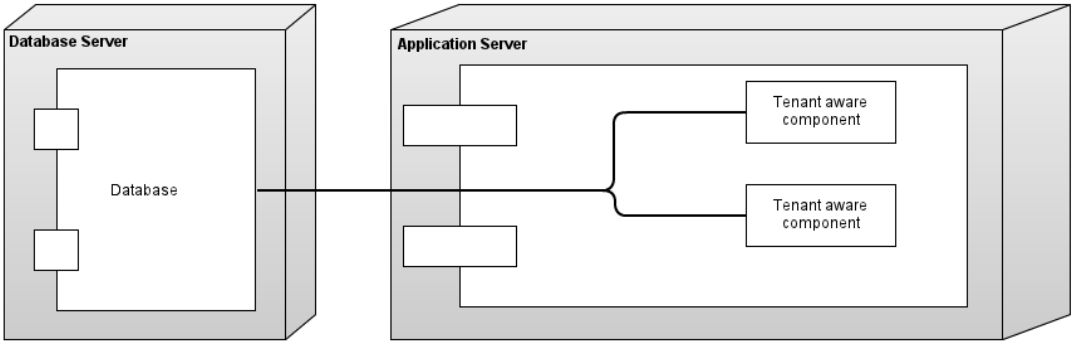


FIGURE A.27: Custom Property Object Pattern: System Model

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

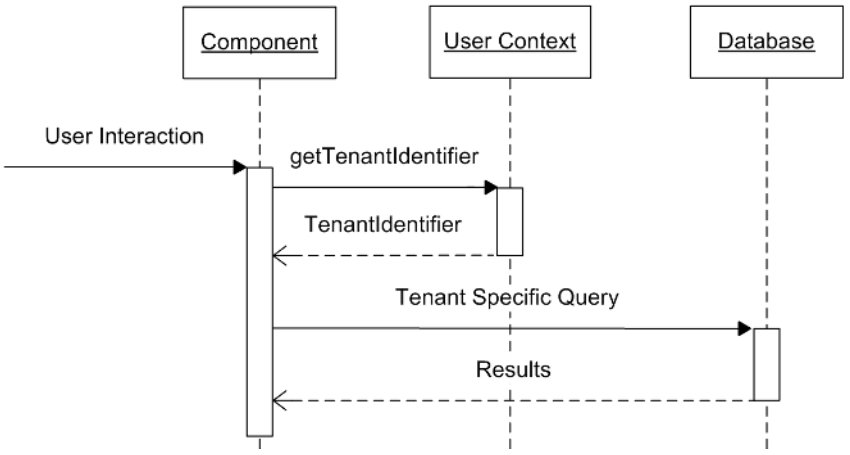


FIGURE A.28: Custom Property Object Pattern:Sequence Diagram

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

Consequences

Security	+ Only a single datasource to secure and maintain - Risk of losing data separation with programming errors
Performance	+ Full resource utilization across all schemas - Loss of optimizations due to lack of type information
Scalability	- No inherent scalability in pattern structure - Requires database system capable of scaling
Maintainability	+ Single static database schema + Custom properties can be handled with generic shared code
Implementation Effort	- Requires adaption of data-access in all components - Custom properties must be handled in all components

TABLE A.16: Consequences of applying the Custom Property Object Pattern

More details on this pattern can be found on page 126.

Appendix B

Publications used in the Structured Mapping Study

This appendix contains a complete list of all papers identified within the structured mapping study performed in Chapter 2. The list is displayed in alphabetical order of first author.

Arya, P. K., V. Venkatesakumar, and S. Palaniswami (2010). "Configurability in SaaS for an electronic contract management application". In: *Proceedings of the International Conference on Networking, VLSI and signal processing (ICNVS)*. ACM, pp. 210–216.

Azeez, A., S. Perera, D. Gamage, R. Linton, P. Siriwardana, D. Leelaratne, S. Weerawarana, and P. Fremantle (2010). "Multi-tenant SOA middleware for cloud computing". In: *Proceedings of the International Conference on Cloud Computing (CLOUD)*. IEEE, pp. 458–465.

Bakshi, K. (2011). "Considerations for cloud data centers: Framework, architecture and adoption". In: *Proceedings of the Aerospace Conference*. IEEE, pp. 1–7.

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

Bezemer, C.-P., A. Zaidman, B. Platzbeecker, T. Hurkmans, and A. t Hart (2010). "Enabling multi-tenancy: An industrial experience report". In: *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE, pp. 1–8.

Cai, H., N. Wang, and M. J. Zhou (2010). "A transparent approach of enabling SaaS multi-tenancy in the cloud". In: *Proceedings of the World Congress on Services*. IEEE, pp. 40–47.

Cai, H., K. Zhang, M. J. Zhou, W. Gong, J. J. Cai, and X. S. Mao (2009). "An end-to-end methodology and toolkit for fine granularity saas-ization". In: *Proceedings of the International Conference on Cloud Computing (CLOUD)*. IEEE, pp. 101–108.

Domingo, E. J., J. T. Niño, A. L. Lemos, M. L. Lemos, R. C. Palacios, and J. M. G. Berbıs (2010). "CLOUDIO: A Cloud Computing-Oriented Multi-tenant Architecture for Business Information Systems". In: *Proceedings of the International Conference on Cloud Computing (CLOUD)*. IEEE, pp. 532–533.

Du, J., X. Gu, and D. S. Reeves (2010). "Highly available component sharing in large-scale multi-tenant cloud systems". In: *Proceedings of the International Symposium on High Performance Distributed Computing*. ACM, pp. 85–94.

Fehling, C., F. Leymann, and R. Mietzner (2010). "A framework for optimized distribution of tenants in cloud applications". In: *Proceedings of the International Conference on Cloud Computing (CLOUD)*. IEEE, pp. 252–259.

Foping, F. S., I. M. Dokas, J. Feehan, and S. Imran (2009). "A new hybrid schema-sharing technique for multitenant applications". In: *Proceedings of the International Conference on Digital Information Management (ICDIM)*. IEEE, pp. 1–6.

Grund, M., M. Schapranow, J. Krueger, J. Schaffner, and A. Bog (2008). "Shared table access pattern analysis for multi-tenant applications". In: *Proceedings of the Symposium on Advanced Management of Information for Globalized Enterprises (AMIGE)*. IEEE, pp. 1–5.

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

Guo, C.-J., W. Sun, Z.-B. Jiang, Y. Huang, B. Gao, and Z.-H. Wang (2011). "Study of Software as a Service Support Platform for Small and Medium Businesses". In: *New Frontiers in Information and Software as Services*. Springer, pp. 1–30.

Jacobs, D., S. Aulbach, et al. (2007). "Ruminations on Multi-Tenant Databases". In: *Fachtagung für Datenbanksysteme in Business, Technologie und Web*, pp. 5–9.

Jiang, X., Y. Zhang, and S. Liu (2010). "A Well-designed SaaS Application Platform Based on Model-driven Approach". In: *Proceedings of the International Conference on Grid and Cooperative Computing (GCC)*. IEEE, pp. 276–281.

Kang, S., S. Kang, and S. Hur (2011). "A Design of the Conceptual Architecture for a Multitenant SaaS Application Platform". In: *Proceedings of the International Conference on Computers, Networks, Systems and Industrial Engineering (CNSI)*. IEEE, pp. 462–467.

Kangarlou, A., D. Xu, U. C. Kozat, P. Padala, B. Lantz, and K. Igarashi (2011). "In-network live snapshot service for recovering virtual infrastructures". In: *IEEE Network* 25.4, pp. 12–19.

Kong, L., Q. Li, and X. Zheng (2010). "A Novel Model Supporting Customization Sharing in SaaS Applications". In: *Proceedings of the International Conference on Multimedia Information Networking and Security (MINES)*. IEEE, pp. 225–229.

Kwok, T., T. Nguyen, and L. Lam (2008). "A software as a service with multi-tenancy support for an electronic contract management application". In: *Proceedings of the International Conference on Services Computing (SCC)*. IEEE, pp. 179–186.

Lee, J. and S. J. Hur (2011). "Level 2 SaaS platform and platform management framework". In: *Proceedings of the International Conference on Advanced Communication Technology (ICACT)*. IEEE, pp. 1177–1180.

Li, X.-Y., Y. Shi, Y. Guo, and W. Ma (2010). "Multi-tenancy based access control in cloud". In: *Proceedings of the International Conference onComputational Intelligence and Software Engineering (CiSE)*. IEEE, pp. 1–4.

Li, X. H., T. C. Liu, Y. Li, and Y. Chen (2008). "SPIN: Service performance isolation infrastructure in multi-tenancy environment". In: *Service-Oriented Computing*. Springer, pp. 649–663.

Lin, H., K. Sun, S. Zhao, and Y. Han (2009). "Feedback-control-based performance regulation for multi-tenant applications". In: *Proceedings of the International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, pp. 134–141.

Mietzner, R., D. Karastoyanova, and F. Leymann (2009). "Business Grid: Combining Web Services and the Grid". In: *Transactions on Petri Nets and Other Models of Concurrency II*. Springer, pp. 136–151.

Mietzner, R., F. Leymann, and M. P. Papazoglou (2008). "Defining composite configurable SaaS application packages using SCA, variability descriptors and multi-tenancy patterns". In: *Proceedings of the International Conference on Internet and Web Applications and Services (ICIW)*. IEEE, pp. 156–161.

Mietzner, R., T. Unger, R. Titze, and F. Leymann (2009). "Combining different multi-tenancy patterns in service-oriented applications". In: *Proceedings of the International Enterprise Distributed Object Computing*. IEEE, pp. 131–140.

Ranchal, R., L. Lilien, B. Bhargava, A. Kim, L. B. Othmane, and M. Kang (2010). "An Approach for Preserving Privacy and Protecting Personally Identifiable Information in Cloud Computing". In: *Unknown Journal*.

Rimal, B. P., E. Choi, and I. Lumb (2010). "A taxonomy, survey, and issues of cloud computing ecosystems". In: *Cloud Computing*. Springer, pp. 21–46.

Schaffner, J., B. Eckart, C. Schwarz, J. Brunnert, D. Jacobs, and A. Zeier (2011). "Towards Analytics-as-a-Service Using an In-Memory Column Database". In: *New Frontiers in Information and Software as Services*. Springer, pp. 257–282.

Sénica, N., C. Teixeira, and J. S. Pinto (2011). "Cloud Computing: A Platform of Services for Services". In: *Enterprise Information Systems*. Springer, pp. 91–100.

Shi, Y., S. Luan, Q. Li, and H. Wang (2009a). "A flexible business process customization framework for SaaS". In: *Proceedings of tje International Conference on Information Engineering (ICIE)*. Vol. 2. IEEE, pp. 350–353.

– (2009b). "A Multi-tenant Oriented Business Process Customization System". In: *Proceedings of the International Conference on New Trends in Information and Service Science (NISS)*. IEEE, pp. 319–324.

Shwartz, L., Y. Diao, and G. Y. Grabarnik (2009). "Multi-tenant solution for it service management: A quantitative study of benefits". In: *Proceedings of the International Symposium on Integrated Network Management*. IEEE, pp. 721–731.

Siddhisena, B., L. Warusawithana, and M. Mendis (2011). "Next generation multi-tenant virtualization cloud computing platform". In: *Proceedings of the International Conference on Advanced Communication Technology (ICACT)*. IEEE, pp. 405–410.

Tang, K., Z. B. Jiang, W. Sun, X. Zhang, and W. S. Dong (2010). "Research on Tenant Placement Based on Business Relations". In: *Proceedings of the International Conference on e-Business Engineering (ICEBE)*. IEEE, pp. 479–483.

Tsai, C.-H., Y. Ruan, S. Sahu, A. Shaikh, and K. G. Shin (2007). "Virtualization-based techniques for enabling multi-tenant management tools". In: *Managing Virtualization of Networks and Services*. Springer, pp. 171–182.

Tsai, W.-T., Q. Shao, and J. Elston (2010). "Prioritizing Service Requests on Cloud with Multi-tenancy". In: *Proceedings of the International Conference one-Business Engineering (ICEBE)*. IEEE, pp. 117–124.

Tsai, W.-T., X. Sun, Q. Shao, and G. Qi (2010). "Two-tier multi-tenancy scaling and load balancing". In: *Proceedings of the International Conference on e-Business Engineering (ICEBE)*. IEEE, pp. 484–489.

Wang, D., Y. Zhang, B. Zhang, and Y. Liu (2009). "Research and Implementation of a New SaaS Service Execution Mechanism with Multi-Tenancy Support". In: *Proceedings of the International Conference on Information Science and Engineering (ICISE)*. IEEE, pp. 336–339.

Wang, X. F. and P. J. Dong (2009). "The multi-tenant data architecture design for the collaboration service system of textile & apparel supply chain". In: *Proceedings of the International Conference on Wireless Communications, Networking and Mobile Computing*. IEEE, pp. 1–4.

Wang, Z. H., C. J. Guo, B. Gao, W. Sun, Z. Zhang, and W. H. An (2008). "A study and performance evaluation of the multi-tenant data tier design patterns for service oriented computing". In: *Proceedings of the International Conference on e-Business Engineering (ICEBE)*. IEEE, pp. 94–101.

Weissman, C. D. and S. Bobrowski (2009). "The design of the force.com multitenant internet application development platform." In: *Proceedings of the SIGMOD Conference*, pp. 889–896.

Wu, M. (2011). "Cloud Computing: Hype or Vision". In: *Applied Informatics and Communication*. Springer, pp. 346–353.

Xu, X. (2012). "From cloud computing to cloud manufacturing". In: *Robotics and computer-integrated manufacturing* 28.1, pp. 75–86.

Zhang, K., Q. Li, and Y. Shi (2011). "Data privacy preservation during schema evolution for multi-tenancy applications in cloud computing". In: *Web Information Systems and Mining*. Springer, pp. 376–383.

Zhang, S. W. and X. P. Wang (2011). "Configuration of Multi-Tenant Applications". In: *Advanced Materials Research* 219, pp. 1182–1185.

Zhang, Y., Z. Wang, B. Gao, C. Guo, W. Sun, and X. Li (2010). "An effective heuristic for on-line tenant placement problem in SaaS". In: *Proceedings of the International Conference on Web Services (ICWS)*. IEEE, pp. 425–432.

Appendix C

List of Acronyms

API Application Programming Interface

ASP Application Service Provider

CMS Content Management System

COTS Commercial Off-The-Shelf

CQRS Command Query Responsibility Separation

CRM Customer Relationship Management

EFG Exploratory Focus Group

ERP Enterprise Resource Planning

ESA Enterprise Software Application

GoF Gang of Four

GUID Globally Unique Identifier

IAAS Infrastructure as a Service

IT Information Technology

MAAM Multi-tenant Architecture Assessment Model

MRQ Main Research Question

MT Multi-Tenancy

MTA Multi-Tenant Architecture

MTSP Multi-Tenant Software Product

MVC Model View Controller

PaaS Platform as a Service

POSA Pattern-Oriented Software Architecture

RQ Research Question

SaaS Software as a Service

SIIA Software & Information Industry Association

SLR Structured Literature Research

SME Small and Medium Enterprise

SMS Systematic Mapping Study

SPEM Software Pattern Evaluation Method

SPL Software Product Line

UML Unified Modeling Language

Appendix D

Personal Publication List

Kabbedijk, J., C. Bezemer, S. Jansen, and A. Zaidman (2014 (In Press)). "Defining Multi-Tenancy: A Structured Mapping Study on the Academic and the Industrial Perspective". In: *Journal of Systems and Software*.

Kabbedijk, J., R. van Donselaar, and S. Jansen (2014). "SPEM: A Software Pattern Evaluation Method". In: *Proceedings of the 5th International Conferences on Pervasive Patterns and Applications (PATTERNS 2014)*, pp. 38–43.

Kabbedijk, J., M. Pors, S. Jansen, and S. Brinkkemper (2014). "Multi-Tenant Architecture Comparison". In: *Proceedings of the 8th European conference on Software Architecture (ECSA'14)*, pp. 202–209.

Kabbedijk, J., S. Jansen, and T. Salfischberger (2014). "Runtime Variability in Online Software Products: A Comparison of Four Patterns". In: *International Journal On Advances in Software* 7.1 and 2, pp. 101–111.

Angeren, J. v., J. Kabbedijk, K. Popp, and S. Jansen (2013). "Managing Software Ecosystems through Partnering". In: *Software Ecosystems: Analyzing and Managing Business Networks in the Software Industry*. Cheltenham, UK: Edward Elgar Publishing. Chap. 5, pp. 85–102.

Kabbedijk, J. and S. Jansen (2013). "Unraveling Ruby Ecosystem Dynamics: A Quantitative Network Analysis". In: *Software Ecosystems: Analyzing and Managing Business Networks in the Software Industry*. Cheltenham, UK: Edward Elgar Publishing. Chap. 15, pp. 322–332.

Kabbedijk, J., T. Salfischberger, and S. Jansen (2013). "Comparing Two Architectural Patterns for Dynamically Adapting Functionality in Online Software Products - Best

Paper Award". In: *Proceedings of the 5th International Conferences on Pervasive Patterns and Applications (PATTERNS 2013)*, pp. 20–25.

Pors, M., L. Blom, J. Kabbedijk, and S. Jansen (2013). *Sharing is Caring - A Decision Support Model for Multi-Tenant Architectures*. Tech. rep. UU-CS-2013-015. Department of Information and Computing Sciences, Utrecht University.

D'souza, A., J. Kabbedijk, D. Seo, S. Jansen, and S. Brinkkemper (2012). "Software-as-a-Service: Implications for Business and Technology in Product Software Companies". In: *Proceedings of the Pacific Asia Conference on Information Systems (PACIS)*. Paper 140.

Kabbedijk, J., M. Galster, and S. Jansen (2012). "Focus Group Report: Evaluating the Consequences of Applying Architectural Patterns". In: *Proceedings of the 17th European conference on Pattern Languages of Programs (EuroPLoP 2012)*.

Kabbedijk, J. and S. Jansen (2012). "The Role of Variability Patterns in Multi-Tenant Business Software". In: *Proceedings of the WICSA/ECSA 2012 Companion Volume*. ACM, pp. 143–146.

Kabbedijk, J., S. Jansen, and S. Brinkkemper (2012). "A Case Study of the Variability Consequences of the CQRS Pattern in Online Business Software". In: *Proceedings of the 17th European Conference on Pattern Languages of Programs*. EuroPLoP '12. Irsee, Germany: ACM, 2:1–2:10.

Nijboer, G., J. Kabbedijk, and S. Jansen (2012). "The Adoption of Software Patterns in the Dutch Software Industry". In: *Proceedings of the 17th European conference on Pattern Languages of Programs (EuroPLoP 2012)*.

Angeren, J. v., J. Kabbedijk, S. Jansen, S. Brinkkemper, and K. Popp (2011). "A Survey of Associate Models used within Large Software Ecosystems". In: *Advances in Software Business*. Books on Demand. Chap. 8, pp. 76–95.

Angeren, J. v., J. Kabbedijk, S. Jansen, and K. Popp (2011). "A Survey of Associate Models used within Large Software Ecosystems". In: *Proceedings of the 3rd International Workshop on Software Ecosystems*. Brussels, pp. 1–13.

Kabbedijk, J. and S. Jansen (2011a). "Steering Insight: An Exploration of the Ruby Software Ecosystem". In: *Proceedings of the 2nd International Conference on Software Business*. Vol. Lecture Notes in Business Information Processing. Springer. Brussels: Springer, pp. 44–55.

– (2011b). "Variability in Multi-Tenant Environments: Architectural Design Patterns from Industry". In: *Proceedings of the 30th international conference on Advances in*

conceptual modeling: recent developments and new directions (ER'11). Springer, pp. 151–160.

Kabbedijk, J., K. Wnuk, B. Regnell, and S. Brinkkemper (2010). "What Decision Characteristics Influence Decision Making in Market-Driven Large-Scale Software Product Line Development?" In: *Proceeding of the 16th International Working Conference on Requirements Engineering: Foundation for Software Quality*. Vol. Proceedings of the Product Line Requirements Engineering and Quality Workshop. Duisburg, Germany, pp. 42–53.

Kabbedijk, J., S. Brinkkemper, S. Jansen, and B. van der Veldt (2009). "Customer Involvement in Requirements Management: Lessons from Mass Market Software Development". In: *Proceedings of the 17th IEEE International Requirements Engineering Conference*. IEEE Publishing. Atlanta, GA: IEEE Publishing, pp. 281–286.

Appendix E

Summary

Enterprise Software Applications (ESAs) have changed significantly over the last decades. A trend is going on in which fewer software products are deployed at the customer's premises and more software is deployed in a central location to be accessed through the internet. Currently the multi-tenancy paradigm is a popular way to offer functionality of a software product through the internet to numerous customers, offering many advantages to software vendors and customers. Major advantages include easy maintenance for software vendors and low total cost of ownership for customers. One of the biggest disadvantages of multi-tenant software, however, is the limited way in which varying requirements can be catered for, without losing one of the aforementioned advantages. Software companies are individually trying to solve this disadvantage and an increasing number of potential solutions are introduced. The plethora of solutions, however, is largely unstructured and trade-offs between the optimally fitting solutions are lacking. Without blueprints for solving challenges in multi-tenancy with well-defined consequences, software producing organizations tend to implement sub-optimal solutions, limiting maintainability, scalability (i.e. growth), and performance of their software products and services.

Problems and solutions in software engineering are typically communicated using software patterns. Patterns are proven solutions to commonly occurring problems, often not only identifying the solution, but also potential uses and consequences. In this dissertation patterns are presented that help implementing variability in

online ESAs. The goal of these patterns is to structure the currently available solutions and document them in a way that allows for comparison of the solutions. Using these patterns, software architects can more easily find the appropriate way to implements variability in online software systems and improve system design.

Main Research Question — How can variability in multi-tenant enterprise software be realized?

This dissertation consists of two major parts. The first part focusses on **variability and multi-tenancy in software systems**. It presents a structured mapping study on multi-tenancy, giving insight in what multi-tenancy is and how it can be used within the domain of software architecture. It also shows the lack of consensus on the definition of multi-tenancy and the consequences of this lack on the communication between academia and industry and research topics related to multi-tenancy. In addition, the first part explains the concept of variability and illustrates how software patterns play an important role in runtime variability in software. The emphasis of this part is on the significance of variability in complying with customer's needs. In addition, patterns are presented contributing to the variability of online ESAs. An example of one of such patterns presented is the Command Query Responsibility Separation (CQRS) pattern, prescribing the strict differentiation between commands and queries in a system implementation. The fragmented nature of this pattern leads to a high number of variability options.

The second part focusses mainly on **selecting the appropriate software patterns in system design**. A collection of twelve multi-tenant architecture patterns is presented, and the Multi-tenant Architecture Assessment Model (MAAM) is introduced. Using this model, software architects can more easily and accurately decide on the most appropriate multi-tenant architecture solutions for their software product. Also additional variability patterns are presented, which are compared to each other based on quality attributes like scalability and performance. Finally, a method is proposed which enables decision makers to compare different patterns in a structured way. In this method, focus groups are used to evaluate the potential solutions.

This dissertation provides insight in the concepts of multi-tenancy, variability and patterns in the domain of online ESAs. All results are gathered from case studies at software companies and evaluated by experts from the software domain. The results support software architects and decision makers in structuring the decision making process by providing a collection of multi-tenant architecture and variability patterns, the MAAM, and a method to setup pattern evaluation and comparison sessions (i.e. SPEM). With these artifacts in hand, software architects can make well-informed decisions and find appropriate patterns for their specific situation, solving the challenges involved in selecting an architecture that support multi-tenant online Enterprise Software Applications. Also, these research results contribute to academia by reporting on numerous case studies in an emerging domain and presenting a vocabulary for further and more extensive research.

Appendix F

Samenvatting

Bedrijfssoftware is aanzienlijk veranderd in de afgelopen decennia. Er is een trend gaande, waarin steeds minder softwareproducten worden geïnstalleerd op locatie bij klanten, en steeds meer software op een centrale locatie wordt geïnstalleerd, die toegankelijk is via het internet. Momenteel is het toepassen van *multi-tenancy* een veelgebruikte manier om een softwareproduct aan te bieden via het internet aan een groot aantal klanten. Dit zorgt voor vele voordelen voor zowel de klant als het softwarebedrijf. Makkelijk onderhoud van de software voor het softwarebedrijf en lage aanschaf- en onderhoudskosten voor klanten zijn een aantal van de belangrijkste voordelen. Echter, een van de grootste nadelen van multi-tenant software is de beperkte manier waarop verschillende programma-eisen aangeboden kunnen worden aan klanten, zonder de eerder genoemde voordelen te verliezen. Verschillende softwarebedrijven zijn individueel bezig om dit nadeel op te lossen en een groeiend aantal mogelijke oplossingen komen momenteel op de markt. De overvloed aan mogelijke oplossingen is echter grotendeels ongestructureerd en een overzicht van de voor- en nadelen van de oplossingen ontbreekt vaak. Zonder duidelijke blauwdrukken voor het oplossen van problemen binnen het domein van multi-tenancy, inclusief duidelijk gevolgen van de oplossing, zullen softwarebedrijven sub-optimale beslissingen blijven maken. Deze sub-optimale beslissingen kunnen leiden tot een slechte onderhoudbaarheid, schaalbaarheid en algehele prestaties van hun softwareproducten en -diensten.

Problemen en oplossingen binnen het domein van softwareontwikkeling worden doorgaans gecommuniceerd door het gebruik van patronen. Patronen zijn bewezen oplossingen voor veelvoorkomende problemen, die vaak niet alleen de oplossing bieden, maar ook mogelijke manieren om de oplossing toe te passen en gevolgen hiervan. In dit proefschrift worden patronen gepresenteerd die softwarebedrijven kunnen ondersteunen bij het implementeren van variabiliteit in online bedrijfssoftware. Het doel van deze patronen is het structureren van de huidige, beschikbare, oplossingen en deze oplossingen vast te leggen op een manier die het mogelijk maakt de verschillende oplossingen te vergelijken. Op deze manier kunnen software-architecten makkelijker variabiliteit implementeren in online softwaresystemen en het algehele software-ontwerp verbeteren.

Hoofdonderzoeksvraag — Hoe kan variabiliteit worden gerealiseerd in multi-tenant bedrijfssoftware?

Dit proefschrift bestaat uit twee hoofddelen. Het eerste deel legt de focus op **variabiliteit en multi-tenancy in softwaresystemen**. Het bevat een 'structured mapping study' op het gebied van multi-tenancy, waar inzicht mee wordt gegeven in wat multi-tenancy is en hoe het gebruikt kan worden binnen het domain van software-architectuur. Tevens wordt het gebrek aan overeenstemming omtrent de definitie van multi-tenancy blootgelegd en de gevolgen hiervan op de communicatie tussen de wetenschap en het bedrijfsleven. Ook wordt in het eerste deel het begrip 'variabiliteit' uitgelegd en wordt geschetst hoe patronen een belangrijke rol spelen in het implementeren van variabiliteit in online softwareproducten. Het eerste deel benadrukt het belang van variabiliteit bij het aanbieden van specifieke klantwensen. Verschillende patronen worden gepresenteerd die helpen bij het implementeren van variabiliteit. Een voorbeeld van een dergelijk patroon is het Command Query Responsibility Separation (CQRS)-patroon, wat een strikte scheiding voorschrijft tussen 'commands' en 'queries' in een systeemimplementatie. Het gefragmenteerde karakter van dit patroon leidt tot meer flexibiliteit in het verschaffen van variabele oplossingen voor klanten.

Het tweede deel legt de focus vooral op het **selecteren van het meest toepasselijke softwarepatroon bij systeemontwerp**. Een verzameling van twaalf

multi-tenant architectuurpatronen wordt gepresenteerd en de Multi-tenant Architecture Assessment Model (MAAM) wordt geïntroduceerd. Door gebruik te maken van dit model kunnen software-architecten makkelijker beslissen omtrent de beste multi-tenant architectuuroplossingen for hun softwareproduct. Ook worden additionele variabiliteitspatronen gepresenteerd, die met elkaar vergeleken worden op basis van kwaliteitsattributen zoals schaalbaarheid en prestatie. Het deel sluit af met een methode die het voor besluitvormers mogelijk maakt zelf verschillende patronen te vergelijken op een gestructureerde manier. In deze methode wordt gebruikt gemaakt van focusgroepen om de potentiele oplossingen te evalueren.

Dit proefschrift geeft inzicht in de concepten *multi-tenancy*, *variabiliteit* en *patronen* binnen het domein van online bedrijfssoftware. Alle resultaten zijn verzameld door middel van 'case studies' bij softwarebedrijven en geëvalueerd door experts binnen het softwaredomein. De resultaten ondersteunen architecten en besluitvormers in het gestructureerd maken van besluiten door het ter beschikking stellen van een verzameling multi-tenant architectuur- en variabiliteitspatronen. Ook MAAM en een methode om patroonevaluaties uit te voeren (i.e. SPEM) dragen hier aan bij. Door gebruik te maken van deze artefacten kunnen softwarearchitecten weloverwogen besluiten nemen en de best passen patronen vinden voor hun specifieke situatie. Ook dragen de resultaten in proefschrift bij aan de wetenschap door het rapporteren van case studies binnen een opkomend domein en een vocabulaire te presenteren voor vervolgonderzoek.

Appendix G

SIKS Dissertation Series

2009

2009-01	Rasa Jurgelenaite (RUN) Symmetric Causal Independence Models.
2009-02	Willem Robert van Hage (VU) Evaluating Ontology-Alignment Techniques.
2009-03	Hans Stol (UvT) A Framework for Evidence-based Policy Making Using IT
2009-04	Josephine Nabukenya (RUN) Improving the Quality of Organisational Policy Making using Collaboration Engineering.
2009-05	Sietse Overbeek (RUN) Bridging Supply and Demand for Knowledge Intensive Tasks.
2009-06	Muhammad Subianto (UU) Understanding Classification.
2009-07	Ronald Poppe (UT) Discriminative Vision-Based Recovery and Recognition of Human Motion.
2009-08	Volker Nannen (VU) Evolutionary Agent-Based Policy Analysis in Dynamic Environments.
2009-09	Benjamin Kanagwa (RUN) Design, Discovery and Construction of Service-oriented Systems.
2009-10	Jan Wielemaker (UVA) Logic programming for knowledge-intensive interactive applications.
2009-11	Alexander Boer (UVA) Legal Theory, Sources of Law & the Semantic Web.
2009-12	Peter Massuthe (TUE, Humboldt-Universitaet zu Berlin) perating Guidelines for Services.
2009-13	Steven de Jong (UM) Fairness in Multi-Agent Systems.
2009-14	Maksym Korotkiy (VU) From ontology-enabled services to service-enabled ontologies
2009-15	Rinke Hoekstra (UVA) Ontology Representation - Design Patterns and Ontologies that Make Sense.
2009-16	Fritz Reul (UvT) New Architectures in Computer Chess.
2009-17	Laurens van der Maaten (UvT)

Feature Extraction from Visual Data.

2009-18 Fabian Groffen (CWI)
Armada, An Evolving Database System.

2009-19 Valentin Robu (CWI)
Modeling Preferences, Strategic Reasoning and Collaboration.

2009-20 Bob van der Vecht (UU)
Adjustable Autonomy: Controling Influences on Decision Making.

2009-21 Stijn Vanderlooy (UM)
Ranking and Reliable Classification.

2009-22 Pavel Serdyukov (UT)
Search For Expertise: Going beyond direct evidence.

2009-23 Peter Hofgesang (VU)
Modelling Web Usage in a Changing Environment.

2009-24 Annerieke Heuvelink (VUA)
Cognitive Models for Training Simulations.

2009-25 Alex van Ballegooij (CWI)
RAM: Array Database Management through Relational Mapping.

2009-26 Fernando Koch (UU)
An Agent-Based Model for the Development of Intelligent Mobile Services.

2009-27 Christian Glahn (OU)
Contextual Support of social Engagement and Reflection on the Web.

2009-28 Sander Evers (UT)
Sensor Data Management with Probabilistic Models.

2009-29 Stanislav Pokraev (UT)
Model-Driven Semantic Integration of Service-Oriented Applications.

2009-30 Marcin Zukowski (CWI)
Balancing vectorized query execution with bandwidth-optimized storage.

2009-31 Sofiya Katrenko (UVA)
A Closer Look at Learning Relations from Text.

2009-32 Rik Farenhorst (VU) and Remco de Boer (VU)
Architectural Knowledge Management: Supporting Architects and Auditors.

2009-33 Khiet Truong (UT)
How Does Real Affect Affect Affect Recognition In Speech?

2009-34 Inge van de Weerd (UU)
Advancing in Software Product Management: An Incremental Method Engineering Approach.

2009-35 Wouter Koelewijn (UL)
Privacy en Politiegegevens; Over geautomatiseerde normatieve informatie-uitwisseling.

2009-36 Marco Kalz (OUN)
Placement Support for Learners in Learning Networks.

2009-37 Hendrik Drachsler (OUN)
Navigation Support for Learners in Informal Learning Networks.

2009-38 Riina Vuorikari (OU)
Tags and self-organisation: a metadata ecology for learning resources in a multilingual context.

2009-39 Christian Stahl (TUE, Humboldt-Universitaet zu Berlin)
Service Substitution – A Behavioral Approach Based on Petri Nets.

2009-40 Stephan Raaijmakers (UvT)
Multinomial Language Learning: Investigations into the Geometry of Language.

2009-41 Igor Berezhnyy (UvT)
Digital Analysis of Paintings.

2009-42 Toine Bogers
Recommender Systems for Social Bookmarking.

2009-43 Virginia Nunes Leal Franqueira (UT)
Finding Multi-step Attacks in Computer Networks using Heuristic Search and Mobile Ambients.

2009-44 Roberto Santana Tapia (UT)
Assessing Business-IT Alignment in Networked Organizations.

2009-45 Jilles Vreeken (UU)
Making Pattern Mining Useful.

2009-46 Loredana Afanasiev (UvA)

Querying XML: Benchmarks and Recursion.

2010

2010-01 Matthijs van Leeuwen (UU)
Patterns that Matter.

2010-02 Ingo Wassink (UT)
Work flows in Life Science.

2010-03 Joost Geurts (CWI)
A Document Engineering Model and Processing Framework for Multimedia documents.

2010-04 Olga Kulyk (UT)
Do You Know What I Know? Situational Awareness of Co-located Teams.

2010-05 Claudia Hauff (UT)
Predicting the Effectiveness of Queries and Retrieval Systems.

2010-06 Sander Bakkes (UvT)
Rapid Adaptation of Video Game AI

2010-07 Wim Fikkert (UT)
Gesture interaction at a Distance.

2010-08 Krzysztof Siewicz (UL)
Towards an Improved Regulatory Framework of Free Software.

2010-09 Hugo Kielman (UL)
A Politiele gegevensverwerking en Privacy, Naar een effectieve waarborging.

2010-10 Rebecca Ong (UL)
Mobile Communication and Protection of Children.

2010-11 Adriaan Ter Mors (TUD)
The world according to MARP: Multi-Agent Route Planning.

2010-12 Susan van den Braak (UU)
Sensemaking software for crime analysis.

2010-13 Gianluigi Folino (RUN)
High Performance Data Mining using Bio-inspired techniques.

2010-14 Sander van Splunter (VU)
Automated Web Service Reconfiguration.

2010-15 Lianne Bodenstaff (UT)
Managing Dependency Relations in Inter-Organizational Models.

2010-16 Sicco Verwer (TUD)
Efficient Identification of Timed Automata, theory and practice.

2010-17 Spyros Kotoulas (VU)
Scalable Discovery of Networked Resources: Algorithms, Infrastructure, Applications.

2010-18 Charlotte Gerritsen (VU)
Caught in the Act: Investigating Crime by Agent-Based Simulation.

2010-19 Henriette Cramer (UvA)
People's Responses to Autonomous and Adaptive Systems.

2010-20 Ivo Swartjes (UT)
Whose Story Is It Anyway? How Improv Informs Agency and Authorship of Emergent Narrative.

2010-21 Harold van Heerde (UT)
Privacy-aware data management by means of data degradation.

2010-22 Michiel Hildebrand (CWI)
End-user Support for Access to. Heterogeneous Linked Data.

2010-23 Bas Steunebrink (UU)
The Logical Structure of Emotions.

2010-24 Dmytro Tykhonov
Designing Generic and Efficient Negotiation Strategies.

2010-25 Zulfiqar Ali Memon (VU)
Modelling Human-Awareness for Ambient Agents: A Human Mindreading Perspective.

2010-26 Ying Zhang (CWI)

XRPC: Efficient Distributed Query Processing on Heterogeneous XQuery Engines.

2010-27 Marten Voulon (UL)
Automatisch contracteren.

2010-28 Arne Koopman (UU)
Characteristic Relational Patterns.

2010-29 Stratos Idreos(CWI)
Database Cracking: Towards Auto-tuning Database Kernels.

2010-30 Marieke van Erp (UvT)
Accessing Natural History - Discoveries in data cleaning, structuring, and retrieval.

2010-31 Victor de Boer (UVA)
Ontology Enrichment from Heterogeneous Sources on the Web.

2010-32 Marcel Hiel (UvT)
An Adaptive Service Oriented Architecture: Automatically solving Interoperability Problems.

2010-33 Robin Aly (UT)
Modeling Representation Uncertainty in Concept-Based Multimedia Retrieval.

2010-34 Teduh Dirgahayu (UT)
Interaction Design in Service Compositions.

2010-35 Dolf Trieschnigg (UT)
Proof of Concept: Concept-based Biomedical Information Retrieval.

2010-36 Jose Janssen (OU)
Paving the Way for Lifelong Learning.

2010-37 Niels Lohmann (TUE)
Correctness of services and their composition.

2010-38 Dirk Fahland (TUE)
From Scenarios to components.

2010-39 Ghazanfar Farooq Siddiqui (VU)
Integrative modeling of emotions in virtual agents.

2010-40 Mark van Assem (VU)
Converting and Integrating Vocabularies for the Semantic Web.

2010-41 Guillaume Chaslot (UM)
Monte-Carlo Tree Search.

2010-42 Sybren de Kinderen (VU)
Needs-driven service bundling in a multi-supplier setting - the computational e3-service approach.

2010-43 Peter van Kranenburg (UU)
A Computational Approach to Content-Based Retrieval of Folk Song Melodies.

2010-44 Pieter Bellekens (TUE)
An Approach towards Context-sensitive and User-adapted Access.

2010-45 Vasilios Andrikopoulos (UvT)
A theory and model for the evolution of software services.

2010-46 Vincent Pijpers (VU)
e3alignment: Exploring Inter-Organizational Business-ICT Alignment.

2010-47 Chen Li (UT)
Mining Process Model Variants: Challenges, Techniques, Examples.

2010-48 Milan Lovric (EUR)
Behavioral Finance and Agent-Based Artificial Markets.

2010-49 Jahn-Takeshi Saito (UM)
Solving difficult game positions.

2010-50 Bouke Huurnink (UVA)
Search in Audiovisual Broadcast Archives.

2010-51 Alia Khairia Amin (CWI)
Understanding and supporting information seeking tasks in multiple sources.

2010-52 Peter-Paul van Maanen (VU)
Adaptive Support for Human-Computer Teams.

2010-53 Edgar Meij (UVA)
Combining Concepts and Language Models for Information Access.

2011

2011-01 Botond Cseke (RUN)
Variational Algorithms for Bayesian Inference in Latent Gaussian Models.

2011-02 Nick Tinnemeier(UU)
Organizing Agent Organizations.

2011-03 Jan Martijn van der Werf (TUE)
Compositional Design and Verification of Component-Based Information Systems.

2011-04 Hado van Hasselt (UU)
Insights in Reinforcement Learning.

2011-05 Base van der Raadt (VU)
Enterprise Architecture Coming of Age - Increasing the Performance of an Emerging Discipline.

2011-06 Yiwen Wang (TUE)
Semantically-Enhanced Recommendations in Cultural Heritage.

2011-07 Yujia Cao (UT)
Multimodal Information Presentation for High Load Human Computer Interaction.

2011-08 Nieske Vergunst (UU)
BDI-based Generation of Robust Task-Oriented Dialogues.

2011-09 Tim de Jong (OU)
Contextualised Mobile Media for Learning.

2011-10 Bart Bogaert (UvT)
Cloud Content Contention.

2011-11 Dhaval Vyas (UT)
Designing for Awareness: An Experience-focused HCI Perspective.

2011-12 Carmen Bratosin (TUE)
Grid Architecture for Distributed Process Mining.

2011-13 Xiaoyu Mao (UvT)
Airport under Control. Multiagent Scheduling for Airport Ground Handling.

2011-14 Milan Lovric (EUR)
Behavioral Finance and Agent-Based Artificial Markets.

2011-15 Marijn Koolen (UvA)
The Meaning of Structure: the Value of Link Evidence for Information Retrieval.

2011-16 Maarten Schadd (UM)
Selective Search in Games of Different Complexity.

2011-17 Jiyin He (UVA)
Exploring Topic Structure: Coherence, Diversity and Relatedness.

2011-18 Mark Ponsen (UM)
Strategic Decision-Making in complex games.

2011-19 Ellen Rusman (OU)
The Mind ' s Eye on Personal Profiles.

2011-20 Qing Gu (VU)
Guiding service-oriented software engineering - A view-based approach.

2011-21 Linda Terlouw (TUD)
Modularization and Specification of Service-Oriented Systems.

2011-22 Junte Zhang (UVA)
System Evaluation of Archival Description and Access.

2011-23 Wouter Weerkamp (UVA)
Finding People and their Utterances in Social Media.

2011-24 Herwin van Welbergen (UT)
Behavior Generation for Interpersonal Coordination with Virtual Humans.

2011-25 Syed Waqar ul Qounain Jaffry (VU)
Analysis and Validation of Models for Trust Dynamics.

2011-26 Matthijs Aart Pontier (VU)
Virtual Agents for Human Communication.

2011-27 Aniel Bhulai (VU)
Dynamic website optimization through autonomous management of design patterns.

2011-28 Rianne Kaptein(UVA)
Effective Focused Retrieval by Exploiting Query Context and Document Structure.

2011-29 Faisal Kamiran (TUE)
Discrimination-aware Classification.

2011-30 Egon van den Broek (UT)
Affective Signal Processing (ASP): Unraveling the mystery of emotions.

2011-31 Ludo Waltman (EUR)
Computational and Game-Theoretic Approaches for Modeling Bounded Rationality.

2011-32 Nees-Jan van Eck (EUR)
Methodological Advances in Bibliometric Mapping of Science.

2011-33 Tom van der Weide (UU)
Arguing to Motivate Decisions.

2011-34 Paolo Turrini (UU)
Strategic Reasoning in Interdependence: Logical and Game-theoretical Investigations.

2011-35 Maaike Harbers (UU)
Explaining Agent Behavior in Virtual Training.

2011-36 Erik van der Spek (UU)
Experiments in serious game design: a cognitive approach.

2011-37 Adriana Burlutiu (RUN)
Machine Learning for Pairwise Data, Applications for Preference Learning.

2011-38 Nyree Lemmens (UM)
Bee-inspired Distributed Optimization.

2011-39 Joost Westra (UU)
Organizing Adaptation using Agents in Serious Games.

2011-40 Viktor Clerc (VU)
Architectural Knowledge Management in Global Software Development.

2011-41 Luan Ibraimi (UT)
Cryptographically Enforced Distributed Data Access Control.

2011-42 Michal Sindlar (UU)
Explaining Behavior through Mental State Attribution.

2011-43 Henk van der Schuur (UU)
Process Improvement through Software Operation Knowledge.

2011-44 Boris Reuderink (UT)
Robust Brain-Computer Interfaces.

2011-45 Herman Stehouwer (UvT)
Statistical Language Models for Alternative Sequence Selection.

2011-46 Beibei Hu (TUD)
Towards Contextualized Information Delivery.

2011-47 Azizi Bin Ab Aziz(VU)
Exploring Computational Models for Intelligent Support of Persons with Depression.

2011-48 Mark Ter Maat (UT)
Response Selection and Turn-taking for a Sensitive Artificial Listening Agent.

2011-49 Andreea Niculescu (UT)
Conversational interfaces for task-oriented spoken dialogues: design aspects influencing interaction quality.

2012

2012-01 Terry Kakeeto (UvT)
Relationship Marketing for SMEs in Uganda.

2012-02 Muhammad Umair(VU)
Adaptivity, emotion, and Rationality in Human and Ambient Agent Models.

2012-03 Adam Vanya (VU)
Supporting Architecture Evolution by Mining Software Repositories.

2012-04 Jurriaan Souer (UU)
Development of Content Management System-based Web Applications.
2012-05 Marijn Plomp (UU)
Maturing Interorganisational Information Systems.
2012-06 Wolfgang Reinhardt (OU)
Awareness Support for Knowledge Workers in Research Networks.
2012-07 Rianne van Lambalgen (VU)
When the Going Gets Tough: Exploring Agent-based Models of Human Performance.
2012-08 Gerben de Vries (UVA)
Kernel Methods for Vessel Trajectories.
2012-09 Ricardo Neisse (UT)
Trust and Privacy Management Support for Context-Aware Service Platforms.
2012-10 David Smits (TUE)
Towards a Generic Distributed Adaptive Hypermedia Environment.
2012-11 J.C.B. Rantham Prabhakara (TUE)
Process Mining in the Large: Preprocessing, Discovery, and Diagnostics.
2012-12 Kees van der Sluijs (TUE)
Model Driven Design and Data Integration in Semantic Web Information Systems.
2012-13 Suleman Shahid (UvT)
Fun and Face: Exploring non-verbal expressions of emotion during playful interactions.
2012-14 Evgeny Knutov(TUE)
Generic Adaptation Framework for Unifying Adaptive Web-based Systems.
2012-15 Natalie van der Wal (VU)
Social Agents. Agent-Based Modelling of Integrated Internal and Social Dynamics.
2012-16 Fiemke Both (VU)
Ambient Agents supporting task execution and depression treatment.
2012-17 Amal Elgammal (UvT)
Towards a Comprehensive Framework for Business Process Compliance.
2012-18 Eltjo Poort (VU)
Improving Solution Architecting Practices.
2012-19 Helen Schonenberg (TUE)
What's Next? Operational Support for Business Process Execution.
2012-20 Ali Bahramisharif (RUN)
Covert Visual Spatial Attention, a Robust Paradigm for Brain-Computer Interfacing.
2012-21 Roberto Cornacchia (TUD)
Querying Sparse Matrices for Information Retrieval.
2012-22 Thijs Vis (UvT)
Intelligence, politie en veiligheidsdienst: verenigbare grootheden?
2012-23 Christian Muehl (UT)
Toward Affective Brain-Computer Interfaces: Exploring the Neurophysiology.
2012-24 Laurens van der Werff (UT)
Evaluation of Noisy Transcripts for Spoken Document Retrieval.
2012-25 Silja Eckartz (UT)
Managing the Business Case Development in Inter-Organizational IT Projects.
2012-26 Emile de Maat (UVA)
Making Sense of Legal Text.
2012-27 Hayrettin Gurkok (UT)
Mind the Sheep! User Experience Evaluation & Brain-Computer Interface Games.
2012-28 Nancy Pascall (UvT)
Engendering Technology Empowering Women.
2012-29 Almer Tigelaar (UT)
Peer-to-Peer Information Retrieval.
2012-30 Alina Pommeranz (TUD)
Designing Human-Centered Systems for Reflective Decision Making.
2012-31 Emily Bagarukayo (RUN)
A Learning by Construction Approach for Higher Order Cognitive Skills Improvement.
2012-32 Wietske Visser (TUD)
Qualitative multi-criteria preference representation and reasoning.

2012-33 Rory Sie (OUN)
Coalitions in Cooperation Networks (COCOON)
2012-34 Pavol Jancura (RUN)
Evolutionary analysis in PPI networks and applications.
2012-35 Evert Haasdijk (VU)
Never Too Old To Learn – On-line Evolution of Controllers in Swarm- and Modular Robotics.
2012-36 Denis Ssebugwawo (RUN)
Analysis and Evaluation of Collaborative Modeling Processes.
2012-37 Agnes Nakakawa (RUN)
A Collaboration Process for Enterprise Architecture Creation.
2012-38 Selmar Smit (VU)
Parameter Tuning and Scientific Testing in Evolutionary Algorithms.
2012-39 Hassan Fatemi (UT)
Risk-aware design of value and coordination networks.
2012-40 Agus Gunawan (UvT)
Information Access for SMEs in Indonesia.
2012-41 Sebastian Kelle (OU)
Game Design Patterns for Learning.
2012-42 Dominique Verpoorten (OU)
Reflection Amplifiers in self-regulated Learning.
2012-44 Anna Tordai (VU)
On Combining Alignment Techniques.
2012-45 Benedikt Kratz (UvT)
A Model and Language for Business-aware Transactions.
2012-46 Simon Carter (UVA)
Exploration and Exploitation of Multilingual Data for Statistical Machine Translation.
2012-47 Manos Tsagkias (UVA)
Mining Social Media: Tracking Content and Predicting Behavior.
2012-48 Jorn Bakker (TUE)
Handling Abrupt Changes in Evolving Time-series Data.
2012-49 Michael Kaisers (UM)
Learning against Learning.
2012-50 Steven van Kervel (TUD)
Ontologogy driven Enterprise Information Systems Engineering.
2012-51 Jeroen de Jong (TUD)
Heuristics in Dynamic Sceduling; a practical framework with a case study in elevator dispatching.

2013

2013-01 Viorel Milea (EUR)
News Analytics for Financial Decision Support.
2013-02 Erietta Liarou (CWI)
MonetDB/DataCell: Leveraging the Column-store Database Technology.
2013-03 Szymon Klarman (VU)
Reasoning with Contexts in Description Logics.
2013-04 Chetan Yadati(TUD)
Coordinating autonomous planning and scheduling.
2013-05 Dulce Pumareja (UT)
Groupware Requirements Evolutions Patterns.
2013-06 Romulo Goncalves(CWI)
The Data Cyclotron: Juggling Data and Queries for a Data Warehouse Audience.
2013-07 Giel van Lankveld (UvT)
Quantifying Individual Player Differences.
2013-08 Robbert-Jan Merk(VU)
Making enemies: cognitive modeling for opponent agents in fighter pilot simulators.

2013-09 Fabio Gori (RUN)
Metagenomic Data Analysis: Computational Methods and Applications.
2013-10 Jeewanie Jayasinghe Arachchige(UvT)
A Unified Modeling Framework for Service Design.
2013-11 Evangelos Pournaras(TUD)
Multi-level Reconfigurable Self-organization in Overlay Services.
2013-12 Marian Razavian(VU)
Knowledge-driven Migration to Services.
2013-13 Mohammad Safiri(UT)
Service Tailoring: User-centric creation of integrated IT-based homecare services.
2013-14 Jafar Tanha (UVA)
Ensemble Approaches to Semi-Supervised Learning Learning.
2013-15 Daniel Hennes (UM)
Multiagent Learning - Dynamic Games and Applications.
2013-16 Eric Kok (UU)
Exploring the practical benefits of argumentation in multi-agent deliberation.
2013-17 Koen Kok (VU)
The PowerMatcher: Smart Coordination for the Smart Electricity Grid.
2013-18 Jeroen Janssens (UvT)
Outlier Selection and One-Class Classification.
2013-19 Renze Steenhuizen (TUD)
Coordinated Multi-Agent Planning and Scheduling.
2013-20 Katja Hofmann (UvA)
Fast and Reliable Online Learning to Rank for Information Retrieval.
2013-21 Sander Wubben (UvT)
Text-to-text generation by monolingual machine translation.
2013-22 Tom Claassen (RUN)
Causal Discovery and Logic.
2013-23 Patricio de Alencar Silva(UvT)
Value Activity Monitoring.
2013-24 Haitham Bou Ammar (UM)
Automated Transfer in Reinforcement Learning.
2013-25 Agnieszka Anna Latoszek-Berendsen (UM)
Intention-based Decision Support.
2013-26 Alireza Zarghami (UT)
Architectural Support for Dynamic Homecare Service Provisioning.
2013-27 Mohammad Huq (UT)
Inference-based Framework Managing Data Provenance.
2013-28 Frans van der Sluis (UT)
When Complexity becomes Interesting: An Inquiry into the Information eXperience.
2013-29 Iwan de Kok (UT)
Listening Heads.
2013-30 Joyce Nakatumba (TUE)
Resource-Aware Business Process Management: Analysis and Support.
2013-31 Dinh Khoa Nguyen (UvT)
Blueprint Model and Language for Engineering Cloud Applications.
2013-32 Kamakshi Rajagopal (OUN)
The role of Networking in a Lifelong Learner's Professional Development.
2013-33 Qi Gao (TUD)
User Modeling and Personalization in the Microblogging Sphere.
2013-34 Kien Tjin-Kam-Jet (UT)
Distributed Deep Web Search.
2013-35 Abdallah El Ali (UvA)
Minimal Mobile Human Computer Interaction.
2013-36 Than Lam Hoang (TUe)
Pattern Mining in Data Streams.
2013-37 Dirk Boerner (OUN)
Ambient Learning Displays.

2013-38 Eelco den Heijer (VU)
Autonomous Evolutionary Art.

2013-39 Joop de Jong (TUD)
A Method for Enterprise Ontology based Design of Enterprise Information Systems.

2013-40 Pim Nijssen (UM)
Monte-Carlo Tree Search for Multi-Player Games.

2013-41 Jochem Liem (UVA)
Supporting the Conceptual Modelling of Dynamic Systems.

2013-42 Léon Planken (TUD)
Algorithms for Simple Temporal Reasoning.

2013-43 Marc Bron (UVA)
Exploration and Contextualization through Interaction and Concepts.

2014

2014-01 Nicola Barile (UU)
Studies in Learning Monotone Models from Data.

2014-02 Fiona Tuliyano (RUN)
Combining System Dynamics with a Domain Modeling Method.

2014-03 Sergio Raul Duarte Torres (UT)
Information Retrieval for Children: Search Behavior and Solutions.

2014-04 Hanna Jochmann-Mannak (UT)
Websites for children: search strategies and interface design.

2014-05 Jurriaan van Reijsen (UU)
Knowledge Perspectives on Advancing Dynamic Capability.

2014-06 Damian Tamburri (VU)
Supporting Networked Software Development.

2014-07 Arya Adriansyah (TUE)
Aligning Observed and Modeled Behavior.

2014-08 Samur Araujo (TUD)
Data Integration over Distributed and Heterogeneous Data Endpoints.

2014-09 Philip Jackson (UvT)
Toward Human-Level Artificial Intelligence.

2014-10 Ivan Salvador Razo Zapata (VU)
Service Value Networks.

2014-11 Janneke van der Zwaan (TUD)
An Empathic Virtual Buddy for Social Support.

2014-12 Willem van Willigen (VU)
Look Ma, No Hands: Aspects of Autonomous Vehicle Control.

2014-13 Arlette van Wissen (VU)
Agent-Based Support for Behavior Change.

2014-14 Yangyang Shi (TUD)
Language Models With Meta-information.

2014-15 Natalya Mogles (VU)
Agent-Based Analysis and Support of Human Functioning in Complex Socio-Technical Systems.

2014-16 Krystyna Milian (VU)
Supporting trial recruitment and design by automatically interpreting eligibility criteria.

2014-17 Kathrin Dentler (VU)
Computing healthcare quality indicators automatically.

2014-18 Mattijs Ghijsen (VU)
Methods and Models for the Design and Study of Dynamic Agent Organizations.

2014-19 Vincius Ramos (TUE)
Adaptive Hypermedia Courses: Qualitative and Quantitative Evaluation and Tool Support.

2014-20 Mena Habib (UT)
Named Entity Extraction and Disambiguation for Informal Text: The Missing Link.

2014-21 Kassidy Clark (TUD)
Negotiation and Monitoring in Open Environments.

2014-22 Marieke Peeters (UU)
Personalized Educational Games - Developing agent-supported scenario-based training.

2014-23 Eleftherios Sidirourgos (UvA/CWI)
Space Efficient Indexes for the Big Data Era.

2014-24 Davide Ceolin (VU)
Trusting Semi-structured Web Data.

2014-25 Martijn Lappenschaar (RUN)
New network models for the analysis of disease interaction.

2014-26 Tim Baarslag (TUD)
What to Bid and When to Stop.

2014-27 Rui Jorge Almeida (EUR)
Conditional Density Models Integrating Fuzzy and Probabilistic Representations of Uncertainty.

2014-28 Anna Chmielowiec (VU)
Decentralized k-Clique Matching.

2014-29 Jaap Kabbedijk (UU)
Variability in Multi-Tenant Enterprise Software.