2013

Variability in Multi-tenant Environments:
Usability versus flexibility in tenant-dependent reporting

Master thesis
Ruben Mijwaart (3779599)
h.r.mijwaart@students.uu.nl
Master Business Informatics
Institute of Information and Computer Science
Utrecht University
8-7-2013

= exact

Universiteit Utrecht

Abstract

Multi-tenant applications enable maximization of economies of scale by offering one shared application and database instance to multiple customers. However, as the shared nature of multi-tenancy makes it impossible to customize by changing source code, tenant-based run-time variability needs to be implemented to satisfy as many customer requirements as possible. Since tenants became responsible for customizing the software, usability is an important aspect to consider when implementing variability. However, there is a lack of well documented techniques on how variability can be implemented and the extent to which the level of variability affects usability is unknown. In order to identify existing techniques for the implementation of variability within the functional area of reporting, a case study on Exact Online – a successful multi-tenant application serving more than 25k customers – was conducted. This resulted in a pattern language constituted by seven variability patterns providing software vendors with reusable solutions for the implementation of tenant-dependent reporting. In order to provide a detailed example on how the Data Web API, REST, Data Middleware, and Abstract Query Builder patterns can be applied, we have implemented a generic query builder web application and a data service within the RESTful OData web API on top of Exact Online. In addition, a usability experiment following the question-suggestion protocol was performed to test the hypothesized negative relationship between flexibility and usability. Results indicate that usability – in both effectiveness and efficiency – is indeed negatively impacted by flexibility. Software vendors that want to extend their potential customer base by increasing the flexibility of their software should be aware of the adverse consequences for usability as a serious decrease in usability causes customers to cancel their subscription. In order to guide SaaS vendors in achieving an optimal balance between flexibility and usability, we propose a variability implementation approach.

Table of Contents

Table of Figures

Table of Tables

1 Introduction

In software development we can distinguish between tailor made- and product software. Tailor made software is developed exclusively for one customer whereas product software is defined as standard software sold to a specific market (Xu & Brinkkemper, 2005). Within the product software industry, we can make another categorization based on the party that is responsible for hosting the software product. On the one hand we have on-premise software which is hosted on the IT infrastructure of the customer. whereas on the other hand there is software as a service, which refers to software that is hosted as a service and accessed by customers through the internet (Chong & Carraro, 2006). Within the product software industry, a shift from on-premise to software as a service (SaaS) can be observed (D'souza, Kabbedijk, Seo, Jansen, & Brinkkemper, 2012). The main incentive for customers to choose SaaS at the expense of on-premise software is the reduction in IT costs by outsourcing the support, maintenance, and purchase of IT infrastructure and software. In addition, the purchase of licenses for on-premise software typically involves high incidental expenses while it is common for SaaS products to have usage based pricing (e.g. per month, user, or transaction). In this economic sense, it becomes easier for customers to switch between software products.

SaaS products can have a multi-tenant or single-tenant architecture while on-premise software is single-tenant per definition. The architectural principles of multi-tenancy enable SaaS providers to exploit the full potential of the economies of scale by offering one shared application and database instance to multiple customers (i.e. tenants). This is contrasting to single-tenant software, in which a dedicated code instance is hosted for each tenant. Even when a separate code instance is hosted for each tenant, hardware can be shared by utilizing virtual machines. However, the utilization rate of the IT infrastructure can be further improved when multiple tenants also share one application and database instance (Bezemer & Zaidman, 2010). With multi-tenant applications, hosting costs per tenant will decrease as the amount of tenants sharing an instance increase.

Besides the reduction in IT infrastructure expenses, also maintenance costs decrease since just one code instance and one database need to be managed. For instance, imagine a SaaS product that needs to be updated with new functionality. In a multi-tenant environment, only one code and database instance need to be updated while single-tenant software requires updating as many instances as there are tenants. However, these advantages due to the created economies of scale do not come for free. Leveraging SaaS by applying a multi-tenancy architecture incurs challenges related to scalability, tenant-isolation and software variability (Bezemer & Zaidman, 2010; Chong & Carraro, 2006). This research addresses the need for variability within multi-tenant environments, empowering the end-user to change software in a simple way, in order to satisfy the requirements specific to their tenant.

Reporting is one of the most important functional areas within business software. Operational data is captured from all supported business processes, but business software is only valuable when they empower organizations to utilize this data within various reporting activities. However, as no business is the same, requirements towards reporting are highly customer specific. For this reason, software variability in the functional area of reporting – i.e. tenant-dependent reporting – should be top priority for SaaS vendors. Due to this priority, this study – which addresses the need for the implementation of 'easy to use' software variability in multi-tenant environments – has a focus on tenant-based reporting variability.

In this study, tenant-dependent reporting refers to any functionality that enables end-users to create customized representations of data such that it creates valuable insights into the business. Examples include the customization of data overviews by applying filters and groupings or creating customized dashboard widgets showing graphs or pivot tables.

1.1. Variability

Software variability refers to the ability of a software system to be changed to make it fit within a specific context (van Gurp, Bosch, & Svahnberg, 2001). Besides variability, also configurability and customizability are terms used to describe the ability of software to be changed to fit within a specific context. Configurability refers to the ability of a system to be changed within a pre-defined scope (e.g. through wizards and configuration settings) whereas customizability typically implies the ability of a system to be changed by adding custom code to extend the application with custom functionality. Configurability and customizability are thus quite different approaches to make an application satisfy requirements specific to one tenant. However, both represent variability since they implement the system's ability to be changed to fit within a specific context (figure 1). During this research, the concepts of variability, configurability and customizability are not used interchangeably. Instead, this research is about variability in multi-tenant environments and both configurability and customizability are used only to denote a specialized implementation of variability.

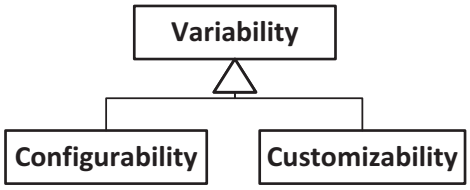

Figure 1 – The concept of software variability

Satisfying the varying requirements of individual tenants by the design of variability within multi-tenant environments is a crucial aspect for SaaS vendors in order to attract a significant number of tenants (Jansen, Houben, & Brinkkemper, 2010; Kabbedijk & Jansen, 2012; Mietzner, Metzger, Leymann, & Pohl, 2009). The more requirements a software product supports, the bigger the market that can be served with a single application instance. Obviously, this principle applies not only to multi-tenant SaaS products, but for software products in general. However, as is illustrated by the four SaaS maturity levels depicted in figure 2, in single-tenant applications customization can also be done by simply changing or adding to the source code since there is only one tenant using this code-base. Software vendors (or their implementation partners) selling on-premise software typically offer this type of customization as an additional service. This single-tenant situation is depicted in the first quadrant, where each tenant is served with its own customized code instance. Another example corresponding to the first situation is variability in software product lines (or families), in which a central architecture can be specialized into concrete static variants (Svahnberg, Van Gurp, & Bosch, 2005). At the second SaaS maturity level, there is still a hosted instance for each tenant, but now the code instances are identical. This increases the maintainability as updating the product to a newer version becomes much simpler, however, as a change in the code now will affect the software of all tenants, run-time variability needs to be implemented. This type of variability enables end-users to change the behavior of a multi-tenant SaaS application during run-time in order to fulfill the requirements specific to their tenant, without changing the code-base and without changing the behavior of the application towards other tenants. At the third maturity level true multi-tenancy is introduced. Now all tenants are served with one code instance. Besides variability, also

tenant isolation becomes a challenge at this level. However, true multi-tenancy introduces the real economies of scale which should make it worth the efforts. The architecture shown at the third level becomes weaker when the amount of tenants grows as it lacks scalability. For this reason, the fourth and highest maturity level introduces a load balancer which divides requests from multiple tenants to a pool of identical code instances.

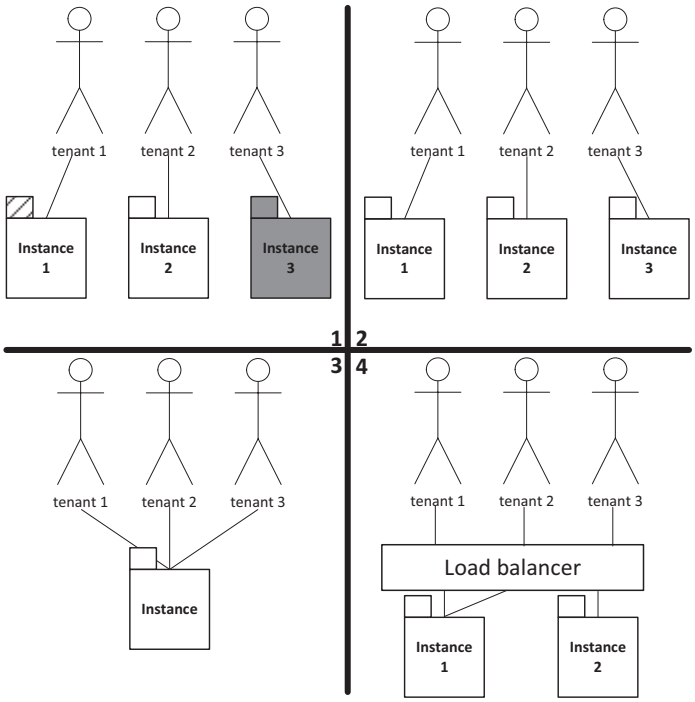

Figure 2 – The SaaS maturity levels by Chong & Carraro (2006)

To summarize, multi-tenant SaaS requires run-time variability to make the application applicable in a broader context (i.e. supporting more customer requirements). However, there is a lack of well documented techniques on how variability can be implemented within multi-tenant environments (Jansen et al., 2010; Kabbedijk & Jansen, 2012).

1.2. Usability

Generally, customization of the old fashioned on-premise software products is performed by the software vendor or implementation partners whereas the customization of SaaS applications – which are particularly interesting for SMEs due to the lower overall costs (Bezemer, Zaidman, Platzbeecker, Hurkmans, & ’t Hart, 2010) – is often done by the customers themselves. This is advantageous for customers as they can avoid consultancy costs. However, this applies only when SaaS vendors succeed to make the customization process as simple as possible (Chong & Carraro, 2006). Obviously, a good design of the configuration interface, the use of configuration wizards, and intuitive screens without causing information overload, all aid the usability of the customization process. Despite a good user interface however, increased variability flexibility will result in an increased complexity as well. This hypothetical

tradeoff is also illustrated by the aforementioned concepts of configurability and customizability. From an end-user perspective, changing an application's behavior by using built-in configurability is much simpler than utilizing the customizability of an application. However, customizability provides much more freedom (i.e. flexibility) than the pre-defined scope implied by configurability. For example, the configuration of the UI (e.g. reflecting corporate branding in the application UI) will generally be a simple task for end-users to complete whereas on the other extreme, the run-time variability introduced by the APEX programming language requires skilled developers to implement customizations on the multi-tenant environment of Force.com (Weissman & Bobrowski, 2009).

1.3. Problem Statement

Based on the need for variability in multi-tenant environments and the importance of variability implementations to have a certain amount of usability, the formal problem statement of this research has been defined:

> *A multi-tenant environment needs to comply with different customer **requirements** in order to maximize its potential customer base. For this reason, variability should be implemented which enables tenants to customize software in a **simple** way. However, there is a lack of knowledge on how **variability** can be implemented and how different implementations impact the **usability** provided to tenants. The absence of this variability leads to increased costs for SaaS vendors.*

1.4. Research Questions

In order to provide scope and guidance for this research while taking the problem statement into account, the following main research question has been defined:

> *How can **tenant-dependent reporting** be implemented in **multi-tenant** environments and how do these different implementations impact the **usability** and **flexibility** provided to end-users?*

While this research addresses the need for variability in multi-tenant environments, the main question introduces the specific focus on the functional area of reporting. When a multi-tenant application enables tenants to create customized reports, this ability can be referred to as tenant-dependent reporting (TDR). In order to answer the main research question, various types of TDR and corresponding implementation techniques need to be identified. In addition, this research addresses the hypothetical tradeoff between the flexibility – which is inherent to software variability – and usability provided to tenants and end-users respectively. To be more specific on how the main question can be answered, the following sub questions have been defined.

RQ1.	*What reporting variability points exist within multi-tenant applications?*
RQ2.	*What variability patterns can be abstracted from the identified variability points?*
RQ3.	*To what extent does the level of variability affect the usability provided by variability points?*

A variability point (VP) can be formally defined as a point in software facilitating a delayed design decision *(Van Gurp et al., 2001).* Variability patterns are specialized software patterns describing solutions to problems related to the implementation of software variability. In other words, variability patterns describe how specific VPs can be implemented. Thus in order to find and document variability patterns, we have to identify VPs first. This is done by means of a case study conducted on Exact Online (EOL), a multi-tenant ERP (Enterprise Resource Planning) system serving more than 25000 tenants at the time of writing. In addition, a prototype VP is developed to bridge the gap between the tenant-data trapped in EOL and the third party reporting tool Google Drive Spreadsheets.

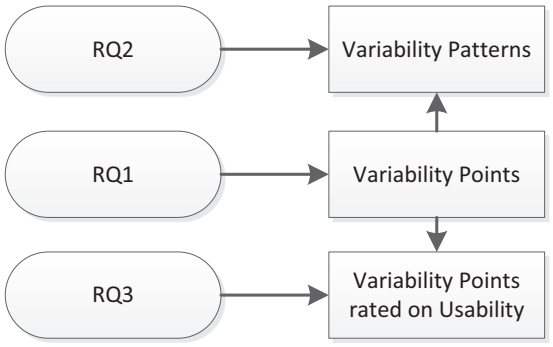

Figure 3 – Questions and deliverables

The third research question concerns the hypothetical tradeoff between flexibility and usability resulting from the general thought that increased freedom leads to increased complexity. Both flexibility and usability are attributes that are perceived as an advantage by customers. When multi-tenant environments offer much flexibility, tenants are able to customize the software to make it suitable for their context. This is also in the best interest of the SaaS provider as their software becomes suitable for a bigger market and more revenue can be generated with one application instance. Usability in the context of software customization refers in a large extent to simplicity. The simpler the process of customizing the application, the higher is the usability perceived by the tenant. The usability of the customization process is way more important for SaaS products than with on-premise solutions as customization in the former is done by the tenant whereas customization in the latter is usually performed by the software vendor extending the standard code base with custom code. In order to determine the usability of the variability implementations, an operationalization of the concept of usability is required. Operationalization is done by splitting up a concept into variables which in their turn are measured by one or more operational definitions that are suitable during a usability evaluation. The thinking-aloud protocol – introduced within the usability field by Clayton Lewis (Lewis, 1982) – is an appropriate method for the assessment of usability (Van den Haak, de Jong, & Schellens, 2004).

However, as the question-suggestion protocol – which is a specialized variant of the thinking-aloud protocol – is more effective in the number of usability problems detected (Grossman, Fitzmaurice, & Attar, 2009), we have applied this approach to assess the usability of the identified variability points. Figure 3 shows how the research questions and deliverables are related.

1.4.1. Scientific Relevance

All related work (see chapter 3 for more details) acknowledges and – to a certain extent – addresses the need for variability in multi-tenant environments. Most of the sources present holistic high level approaches for the realization of variable (i.e. configurable or customizable) multi-tenant environments. These holistic approaches can be categorized into architectures specifically focused on variability, and architectures addressing all major challenges (i.e. scalability, tenant-isolation, and variability) of the realization of multi-tenancy. Furthermore, we have encountered a model for the assessment of the variability maturity of multi-tenant applications and various sources that elaborate on the concept of variability by identifying different variability types from various perspectives. The potential role software patterns can play in both the research and practice of variability is emphasized by Kabbedijk & Jansen (2012). At the time of writing, there is only one source that explicitly provides variability patterns (Kabbedijk & Jansen, 2011). This research addresses the need for variability patterns by performing a case study on the successful industrial multi-tenant environment of Exact Online. As a novelty within the research area of variability, the (candidate) patterns observed during this study are evaluated on the relative usability and flexibility they provide to end-users and tenants respectively. By addressing the hypothetical tradeoff between variability usability and – flexibility, we do not only extend the literature with previously undocumented variability patterns, but also address the need for simplicity within the customization process.

2 Research Approach

2.1. Design Science

The deliverables of this research are aimed at addressing the business needs as they provide guidance for SaaS providers willing to implement software variability in general, and tenant-dependent reporting in particular. All the actions performed to reach this end, together with the resulting sub deliverables and obtained insights, are described in order to add value to the current body of knowledge in the fields of multi-tenancy, variability and software patterns. As the goal of this research is to address relevant business problems while adding value to the current body of knowledge, this research has the characteristics to fit with the design science research paradigm. Therefore, this research is conducted following the guidelines proposed by Hevner, March, Park, & Ram (2004). In 2010, Hevner & Chatterjee extended the IS research framework with an overlay of three research cycles and a design science research checklist. This section elaborates on how the guidelines of the extended version of the framework are applied within the context of this research.

2.1.1. Design Science Guidelines

The seven guidelines – proposed by Hevner et al. (2004) – need to be addressed in order to conduct and evaluate good design science research. This research describes the guidelines and the way they are addressed during this research.

Design as an Artifact

"Design research must produce a viable artifact in the form of a construct, a model, a method, or an instantiation."

During this research, a prototype reporting variability point (VP) is developed for EOL. In addition, existing reporting VPs are identified during a case study on EOL. Both the prototype and the existing VPs serve as input for a pattern language, which is an artifact describing how to solve various problems related to reporting variability in multi-tenant environments. In addition, both the prototype and existing VPs are used for a usability evaluation resulting in a model showing how software variability impacts the usability provided to end-users. These artifacts – together with all the experiences and insights gained during the research process – provide guidance to SaaS providers that want to implement (reporting) variability in their multi-tenant environment.

Problem

"The objective of design science research is to develop technology-based solutions to important and relevant business problems."

The business problem solved by this research concerns the need for variability in multi-tenant environments – in order to maximize the potential customer base that can be served with one instance – and the need to keep the resulting customization process – as this is performed by the tenants – as simple as possible. The owners of the business problem are SaaS providers willing to implement tenant-

dependent reporting. For this reason, these SaaS providers are the target audience of this design science research. However, despite the fact that this research is focused on reporting, knowledge gained other than the actual patterns are relevant for the implementation of variability in general. This broadens the audience to SaaS providers willing to implement variability. In fact, considering the criticality of variability in multi-tenant environments, all SaaS providers owning products with a lack of variability cope with the business problem addressed by this research. The deliverables of this research – as described in the previous sub section – are aimed to help SaaS providers solving these relevant business problems.

Design Evaluation

"The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods."

During the entire research process, (sub) deliverables are evaluated by various means as part of the case study. The (candidate) patterns found during the case study of Exact Online are cross-validated with the current body of knowledge on software patterns. The new candidate pattern designed during this research is evaluated by developing a prototype within the context of the business problem. Both the prototype and the (candidate) variability patterns found during the case study are evaluated by tenants (Customers of Exact Online) in terms of their perceived flexibility and usability.

Research Contributions

"Effective design science research must provide clear and verifiable contributions in the areas of the design artifact, design foundations, and/or design methodologies."

For multi-tenant SaaS to be successful, both variability (Jansen et al., 2010; Kwok, Nguyen, & Lam, 2008; Mietzner et al., 2009) and its usability (Chong & Carraro, 2006) need to be properly addressed. However, there is a lack of well documented techniques for the implementation of variability in multi-tenant SaaS (Kabbedijk & Jansen, 2011). (Kabbedijk & Jansen, 2012) describe how variability patterns can play an important role in addressing this need. Following their approach, multiple patterns for the implementation of tenant-dependent reporting in multi-tenant environments are identified, described, and analyzed in terms of the flexibility and usability they provide to customers. This is an enrichment for the relatively young and unexplored area of variability patterns in multi-tenant SaaS. In addition, one of the patterns is implemented as a prototype during the case study. This results in extensive knowledge about the application of a novel architectural design in the industrial practice.

Research Rigor

"Design science research relies upon the application of rigorous methods in both the construction and evaluation of the design artifact."

Various means are utilized to guarantee the rigorousness of both the construction and evaluation of the design artifact. The design process follows the guidelines provided by the proven design science paradigm consisting of the seven design science guidelines (Hevner et al., 2004), the IS research framework extended with the design science research cycles, and the design science checklist as described by Hevner & Chatterjee (2010). We conduct a multi-faceted case study on Exact Online, a successful multi-tenant application, in order to;

- discover (candidate) variability patterns for the implementation of tenant-dependent reporting;
- implement a novel variability pattern as a prototype within the context of a multi-tenant environment;
- assess the usability of the variability implemented by the identified patterns (including the developed prototype) by performing usability evaluations.

In order to conduct a rigorous case study, a case study protocol is designed following the guidelines provided by Yin (2008), and Runeson & Höst (2009). In this protocol the objective, the case that is being studied, theories and resulting research questions, data collection methods, and selection strategies are elaborated. Details on the case study protocol are provided in the case study design section. The usability evaluations are conducted according the question-suggestion protocol (Grossman et al., 2009).

Design as a Search Process

"The search for an effective artifact requires utilizing available means to reach desired ends while satisfying laws in the problem environment."

High quality scientific literature is widely available through subscriptions of the Utrecht University which enables the researchers to perform proper literature studies. In addition, a great part of the knowledge is obtained through the comprehensive multi-faceted case study. These various sources of knowledge and the continuous interaction with the problem environment facilitate the evolvement towards a model capable of solving the business problem.

Communication of Research

"Design science research must be presented effectively to both technology-oriented and management-oriented audiences."

The results of this design science research are presented at the case company to both product managers and technical oriented audience. In addition, the results are presented to academics/researchers during a graduation session at Utrecht University. The thesis resulting from this research is available to all researchers and students of the Utrecht University and a scientific paper is written with the intention to be submitted for publication.

2.1.2. Design Science Research Cycles

The information systems research framework combines design science with the behavioral science paradigm (Hevner et al., 2004). This combination is also described as the complementary research cycle where the artifacts resulting from design science provide utility for the problem domain, and the behavioral science seeks to provide truth by developing hypothesis and empirically justification of theories (Hevner & Chatterjee, 2010). In figure 4, the properties of this research are mapped into the IS research framework.

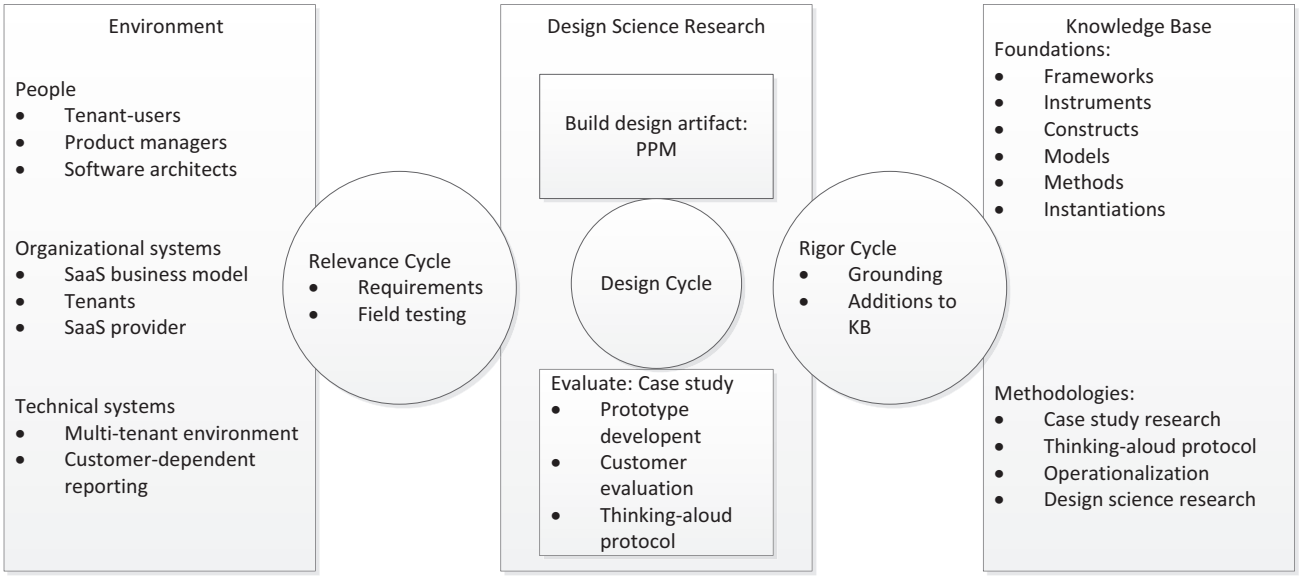

Figure 4 – The IS research framework extended with research cycles

The model is built around three main concepts namely, the environment, IS research and the knowledge base. The problem space is defined by the environment which consists of people, organizational systems and technological systems. The definition of the environment or problem space should be linked with research activities to ensure that the business needs are addressed (i.e. ensure research relevance). This research is defined by the organizational goal of implementing customer dependent-reporting within a multi-tenant environment, their technical implementation of the SaaS model, implicating the involvement of tenants (a customer organization using the multi-tenant product) and their tenant-users (employees working at the tenant). Product managers and software architects or other technical experts have to decide on the actual solution for the implementation of tenant-dependent reporting. Design science research addresses the business needs by constructing a design artifact – the Pattern Positioning Model – and evaluating its sub-deliverables by prototype development and usability evaluations both performed in the practice of the problem domain. The knowledge base represents the prior research in IS – a.k.a. the foundations – and the methodologies which are applied for evaluating research results.

The relevance cycle provides the design science research with requirements and acceptance criteria from the problem environment. The iterations of the relevant cycle involve field testing of the design artifact. The outcome of this evaluation can concern the requirements (i.e. the design artifact did not address the actual problem or opportunity from the problem environment) or the design artifact (i.e. the requirements are set right but are not sufficiently addressed by the design artifact). To ensure the innovation and rigorousness of the design science research, the rigor cycle provides the foundations and methodologies from the knowledge base. The rigor cycle addresses the research contribution that should be made to the current body of knowledge and the theories and methods used for the construction and evaluation of the design artifact.

The design cycle utilizes the design and evaluation theories from the rigor cycle to create the research deliverables and evaluate them against the requirements from the relevance cycle. The eight questions of the design science research checklist can be mapped to the three design research cycles (Hevner & Chatterjee, 2010). In the table below, the number of the question indicates the chronological order in which the questions should be addressed and the research cycle in which a question is addressed is provided as well. The questions form a checklist that is used to make sure that the key aspects of design science are addressed and can be used to keep track of the progress of this design science research.

Table 1 – Design science research checklist mapped to the research cycles

#	Research cycle	Question
1	Relevance	What is the research question (design requirements)?
2	Design	What is the artifact? How is the artifact represented?
3	Design	What design processes (search heuristics) will be used to build the artifact?
4	Rigor	How are the artifact and the design processes grounded by the knowledge base? What, if any, theories support the artifact design and the design process?
5	Design	What evaluations are performed during the internal design cycles? What design improvements are identified during each design cycle?
6	Relevance	How is the artifact introduced into the application environment and how is it field-tested? What metrics are used to demonstrate artifact utility and improvement over previous artifacts?
7	Rigor	What new knowledge is added to the knowledge base and in what form?
8	Relevance	Has the research question been satisfactory addressed?

2.2. Case Study Design

Case study research is the overarching data collection method as all other data collection methods (e.g. thinking-aloud sessions, prototype development, and expert interviews) are applied within the context of the case study. The main data collection objectives include a collection of variability points (VPs), a pattern language containing variability patterns related to those VPs, and usability measures of the VPs. The next subsections elaborate on how these case study objectives are achieved and how we addressed the threats to research validity and reliability.

2.2.1. Identification of Variability Points and Patterns

Due to the relative novelty of multi-tenancy, there are little patterns for external run-time tenant-based variability described in literature. Not to mention the lack of more specific patterns aimed at tenant-dependent reporting. Since patterns are not artificially designed artifacts, but recurring and proven solutions to a common problem, patterns need to be observed in a representative case. For this reason, performing a case study on a successful multi-tenant SaaS application is a suitable data collection method for answering the first question. The case study is performed at Exact (the case company) on their multi-tenant ERP application called Exact Online (the case). The case study related to the first

research question is of an exploratory nature as we aim to find novel patterns by observing an industrial multi-tenant application. Exploratory case studies are aimed at seeking new insights and generating ideas or hypothesis (Robson, 2002). It is not possible to find patterns by observing one case – or one multi-tenant environment – since a pattern is only a pattern when a certain degree of recurrence is observed. To have generalizable results, multiple case studies on different multi-tenant environments have to be conducted. However, the observed solutions for tenant-dependent reporting during one case study can be abstracted and documented as so called candidate patterns (Antoniol, Fiutem, & Cristoforetti, 1998). This thought corresponds with seeking new insights and hypothesis which makes the first case study an exploratory one. In addition to the identification of variability patterns, a novel solution for tenant-dependent reporting is developed within Exact Online, serving as a proof of concept. When this prototype is proven successful, this solution can be abstracted into candidate patterns as well.

2.2.2. Usability Assessment

The third research question concerns the presumed tradeoff between the flexibility resulting from the implementation of software variability and the usability the software provides to end-users. As mentioned in the previous section, a case study is conducted on Exact Online, in order to find variability points which serve as input for the pattern language. However, these VPs are also used to address the third research question by measuring and comparing the usability of VPs, providing a varying degree of flexibility. In order to measure the VPs usability, evaluation sessions are performed using the question-suggestion protocol (Grossman et al., 2009). A thinking-aloud session involves participants that perform tasks using the selected variability points. While performing the assignment, users have to think aloud which enables the observer to discover the various difficulties that might arise. In addition to the collection of qualitative data, the thinking-aloud sessions also enable us to collect quantitative data such as the amount of time participants need to complete tasks, time between inputs, or the number of mouse clicks per task. Details on the usability evaluation protocol and usability metrics are provided in later chapters.

2.2.3. Embedded Single Case Design

A case study is considered to be embedded when the study involves multiple units of analysis within one case (and automatically one context) whereas a study in which the case represents also the one and only unit of analysis is referred to as a holistic case study (Yin, 2008). Figure 5 depicts an abstract embedded single case study on the left and the matching case study design of this research on the right. This research is built around the variability points that are identified during a case study on Exact Online (EOL). They serve as input for the pattern language and are used in the usability evaluation. These variability points are thus considered to be the units of analysis. In addition to the existing VPs, the prototype is implemented within the context of the EOL.

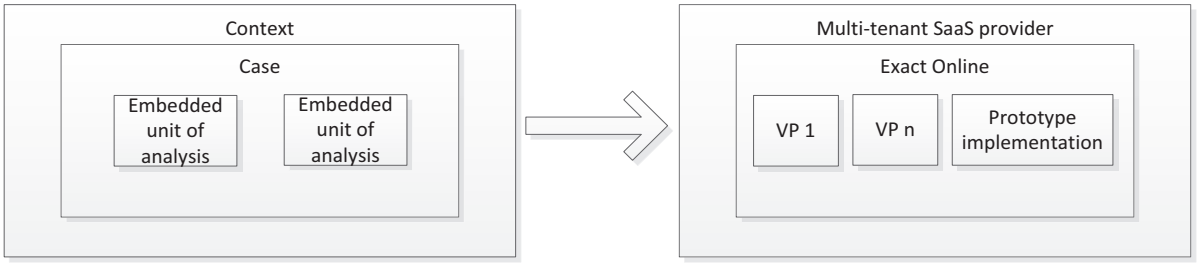

Figure 5 – Embedded single case study at Exact

Besides holistic or embedded, a choice between a single-case and multiple-case design has to be made. The results of a multiple-case design are typically more generalizable than those of a single-case design. Yin describes five rationales which serve as major reasons to choose for a single-case design. When at least one of these rationales is applicable to the case study at hand, the single-case study design is considered to be a valid design decision. One rationale is the *representative* or *typical* case. Exact Online represents a typical multi-tenant SaaS application and results can therefore be assumed to be informative about the average multi-tenant SaaS application. The next section explains how the external validity – i.e. generalizability – is addressed within this case study.

2.2.4. Validity

The quality of a research design can be assessed by four tests each addressing a different aspect of research validity. This section describes the four validity tests and how they are addressed within the case study design. The case study is designed to provide answers to two research questions resulting in one exploratory and one explanatory case study objective. Depending on the type of objective (e.g. explanatory or exploratory), various case study tactics can be applied to address the four validity tests (Runeson & Höst, 2009; Yin, 2008).

Construct Validity

The construct validity test concerns the establishment of correct operational measures for the concepts of interest – a traditional weakness within case study research (Yin, 2008). During this research we measure the usability that variability points provide to end-users. To ensure construct validity, the concept of usability needs to be operationalized by splitting it up into relevant and more concrete variables for which multiple 'operational' definitions are determined. Consequently, the operational definitions are used during the customer-evaluations in order to determine usability of pattern implementations. For the operationalization of usability, a literature study is performed which is elaborated in the theoretical background. A similar approach is applied for the concepts of variability (flexibility) and variability patterns which are required for answering the first and second research question.

Since qualitative data – used for the identification of variability patterns – is less precise than quantitative data, it is important to use multiple sources of evidence (Yin, 2008). This tactic of taking different angles towards the unit of analysis is also referred to as data source triangulation (Stake, 1995). For the identification of variability patterns within the case – Exact Online – we have access to:

- a test account enabling us to observe external variability via the application's GUI;
- experts like software engineers and product managers that can be interviewed;
- the source code of EOL and;
- technical documentation.

Another tactic to ensure construct validity is to create a chain of evidence which explicitly links the research questions, the collected data, and the drawn conclusions (Yin, 2008). This enables the readers of this research to trace the steps from both the research questions to the conclusions and vice versa.
The last case study tactic that we apply to ensure construct validity is to have the case study results reviewed by key informants. Qualitative data resulting from thinking-aloud sessions during usability evaluations can be reviewed by the observed end-users whereas observed and documented variability patterns can be reviewed by a senior software engineer working on Exact Online.

Internal Validity

This type of validity is in particular relevant for case study objectives involving a causal relationship. Internal validity concerns the risk that when we investigate a causal relationship between X and Y, there can be a factor Z which actually did affect X (Runeson & Höst, 2009). When researchers are not aware of Z, internally invalid conclusions might be drawn.
This research is largely of an exploratory nature as the observing and implementation of variability patterns cover a significant part of the case study. However, the assessment of pattern usability involves the hypothetical tradeoff between usability and flexibility. When we investigate whether flexibility affects usability we need to address the risk that another factor actually did affect usability in order to ensure internal validity. One strategy to counteract this risk is by relying on the theoretical propositions that led to a case study (Yin, 2008). For this research, the proposition involves a negative relationship between the flexibility provided to tenants and the usability provided to end-users:

> *Variability points providing high flexibility to tenants provide lower usability to end-users compared to variability points providing lower flexibility.*

The proposition brings the focus to relevant data, helps us to organize the entire case study, and in addition, a proposition helps us to define rival explanations that need to be examined – a second strategy to secure internal validity (Yin, 2008). Two serious rival explanations for an observed change in usability (i.e. high flexibility variability points having lower usability compared to low flexibility variability points) can be the:

- level of experience an end-user has with similar software functionality or;
- the differences in the GUI of the various variability points instead of their difference in flexibility provided.

When we perform customer-evaluations, participants with a high level of experience with similar functionality might have fewer troubles with performing certain tasks than participants with a lower amount of experience. The difference in the GUI amongst variability points is an example of a direct rival,

since the total effect (usability change) is caused by a rival factor (the GUI) instead of the target factor (flexibility). Another plausible rival explanation can be the commingled rival, which refers to the scenario in which both the target and rival factor caused a change in usability. Both rival types are examples of the real-life rivals listed by Yin (2008) and should be considered during data collection and data analysis. The case study should be designed to collect evidence supporting the rival explanations as well. For this reason, in addition to the flexibility and usability, we also collect data about the level of experience of participants and the quality characteristics of the GUIs of the different variability points.

Note that the operationalization of the concept of usability should be done in such a way, that we end up with a working definition that – as far as possible – excludes quality characteristics of the GUI. Usability generally refers to a relative measure of the extent to which a (part of a) software product enables users to achieve specified goals (Seffah, Donyaee, Kline, & Padda, 2006). The implementation of a variability pattern results in a piece of functionality (variability point) which enables the end-user to customize the application to make it fit within the context of its tenant. When we refer to the usability of such a variability point, we refer to aspects of usability that are inherent to the variability pattern. For example, imagine a variability pattern describing a solution for the implementation of a query builder. This pattern describes how data-source specific query constructs (e.g. MS SQL, RESTful API) can be abstracted into general source-independent constructs which enable end-users to create tenant-dependent data overviews. This pattern determines how an end-user can create customized reports and this process can be more or less complex, efficient, understandable, effective, etc., than other patterns describing a solution for tenant-dependent reporting. The GUI that is built on top of this functionality has an undeniable impact on the usability perceived by the end-user while it is entirely independent of the pattern of interest. To summarize, we developed an unambiguous operationalization of pattern usability and counteract the recognized possibility of the GUI to impact the perceived usability by setting up a rival explanation and including GUI quality characteristics in the data collection and analysis.

Real-life rivals may also become apparent during data collection. When this happens, the case study protocol is extended with actions addressing the new rival explanation. The more rival explanations can be addressed and rejected, the higher the confidence that we can place in the results of this study.

External Validity

The extent to which results are generalizable beyond the case study is addressed by the external validity test. In contrast to survey research, which relies on statistical generalization, case study research relies on analytical generalization in which investigators aim to generalize results to some broader theory (Yin, 2008). Yin proposed two case study tactics addressing the external validity; the use of theory in single case studies; and the use of replication logic in a multiple-case study design. Since the case study has an embedded single case design which comprises a single-case study on Exact Online and an embedded multiple-case study on the various variability points (identified in the single case study on Exact Online), we apply both tactics. One case study objective is to identify variability patterns addressing problems related to tenant-dependent reporting. The main challenge with discovering patterns is in fact a generalizability issue as a pattern is only a pattern when it has a certain recurrence. Since this study includes only one case in which patterns are observed, the easiest way to solve this problem is by identifying *candidate patterns* which do not differ from patterns apart from their lack of proven recurrence. By identifying candidate patterns we do not generalize to other cases, instead, we generalize

the findings to theory. More case studies (observations) are required to prove that a candidate pattern (which in this sense can be seen as a hypothesis) is actually a true pattern. As another means to increase generalizability we check observed candidate patterns against existing pattern literature. When an observed pattern was already documented in literature, we do not need to perform additional case studies to prove its recurrence.

Our second case study objective is to determine the relative usability of the observed variability points. The embedded part of the case study involves a small case study on each implemented variability pattern. By performing multiple case studies, we increase the external validity of the results. We apply theoretical replication (Yin, 2008) as we perform multiple cases in which we expect contrasting results as predicted in the proposition. Another means to increase external validity is to perform multiple usability evaluations on each variability point.

One of the rationales for selecting a single case is when it is considered to be a *representative case*. Both from a technical perspective – Exact Online is configurable, multi-tenant efficient and scalable which classifies it within the highest SaaS maturity level (Chong & Carraro, 2006) – and from a business perspective – Exact Online is market leader of online business software in the Netherlands, has more than 25k of typical SaaS customers (SME) spread over multiple countries – this case is considered as a true multi-tenant application that plays a significant role in the industry. Having a *representative case* increases the generalizability of the results (Yin, 2008).

Reliability

When the complete research procedure would be repeated on the same case, but by another researcher, the findings and conclusions should be the same. In other words, the goal of the reliability test is to minimize the errors and bias in a study (Yin, 2008). We address the reliability of this study by developing a case study protocol and maintaining a case study database. The case study protocol is a detailed plan on case study execution and includes data collection procedures, case study questions and propositions, and guidelines on the evaluation of the findings. The case study database is a collection of the raw data or evidence collected. This enables other researchers to redo the analysis which would be impossible if they only had the case study report available. Besides that they heavily increase research reliability, the case study protocol and database are perfect means to facilitate a chain of evidence, explicitly linking the research questions to the evidence and the conclusions drawn (see figure 6).

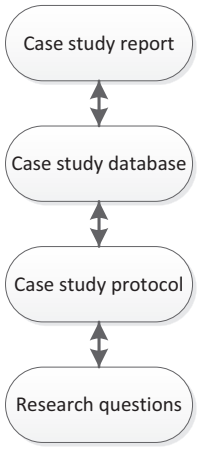

Figure 6 – Chain of evidence Yin (2008)

2.3. Literature Study

In order to provide both an overview of the previous scientific work related to this research and a theoretical background on the main subjects this research, a literature study is performed. The main topics of this multi-disciplinary research include multi-tenancy, software variability, software patterns usability, and business intelligence. The theoretical background describes how this research relates with each topic, defines the important concepts, and describes the relations between these concepts in a summarizing conceptual framework.

The scientific sources elaborated in the related work are gathered via Google Scholar using the combinations of variability (or equivalents e.g. customizability, configurability) and multi-tenancy (or equivalents e.g. SaaS, web applications, and online business software). After a forward and backward reference search a set of 50 papers was collected of which only 34 had sufficient relevance and quality. Besides the related work section, the other sections of the theoretical background use an additional 32 sources to explain concepts relevant to this research.

2.4. Research Process

To summarize the research approach, design science is the research framework that provides the guidelines for understanding, executing and evaluating this research whereas case study research is the overarching data collection method for which guidelines by Yin (2008) and Runeson & Höst (2009) are followed in order to ensure valid and reliable results. During the multi-faceted case study, variability points are identified and abstracted into patterns, a prototype is developed, and the variability points are evaluated in terms of usability by performing usability evaluations using the thinking-aloud protocol (Lewis, 1982). The high level research process – consisting of the performed research actions and their resulting deliverables – is modeled in the Product Deliverable Diagram (PDD) below. A PDD can be used to show the relation between activities and resulting deliverables (van de Weerd & Brinkkemper, 2008). The activities are shown on the left side of the diagram and modeled conform the UML activity diagram standard. The deliverables resulting from this research are modeled as concepts and shown on the right side of the PDD modeled conform the UML class diagram standard. The activity table and concept table below provide more detailed information about the actions performed during this research and the resulting deliverables respectively.

Table 2 – A description of the research activities

Activity	Sub activity	Description
Identify VPs	Build prototype	A PROTOTYPE reporting variability point is designed and implemented within the context of EOL.
	Identify VPs in EOL	Existing VARIABILITY POINTs are identified by performing a case study on EOL.
Identify Patterns	Identify variability patterns	Based on these VARIABILITY POINTs, reusable solutions are abstracted and documented as variability PATTERNS.
	Write pattern language	In order to increase the reusability and simplicity of the identified PATTERNS, they are written as part of a PATTERN LANGUAGE, together solving problems related to tenant-dependent reporting.
Study usability versus flexibility	Define evaluation protocol	A usability evaluation PROTOCOL is written. The concept of usability needs to be operationalized, resulting in metrics for which data needs to be collected. In addition, criteria for participant selection need to be defined as well as the tasks that are conducted and the VPs that are used by participants to complete these tasks.
	Perform usability evaluation	Based on the PROTOCOL, the usability evaluations are conducted resulting in EVALUATION DATA.
	Analyze data	Based on this data – both qualitative and quantitative – an ANALYSIS is

	performed which should give insights in the relationship between flexibility and usability.

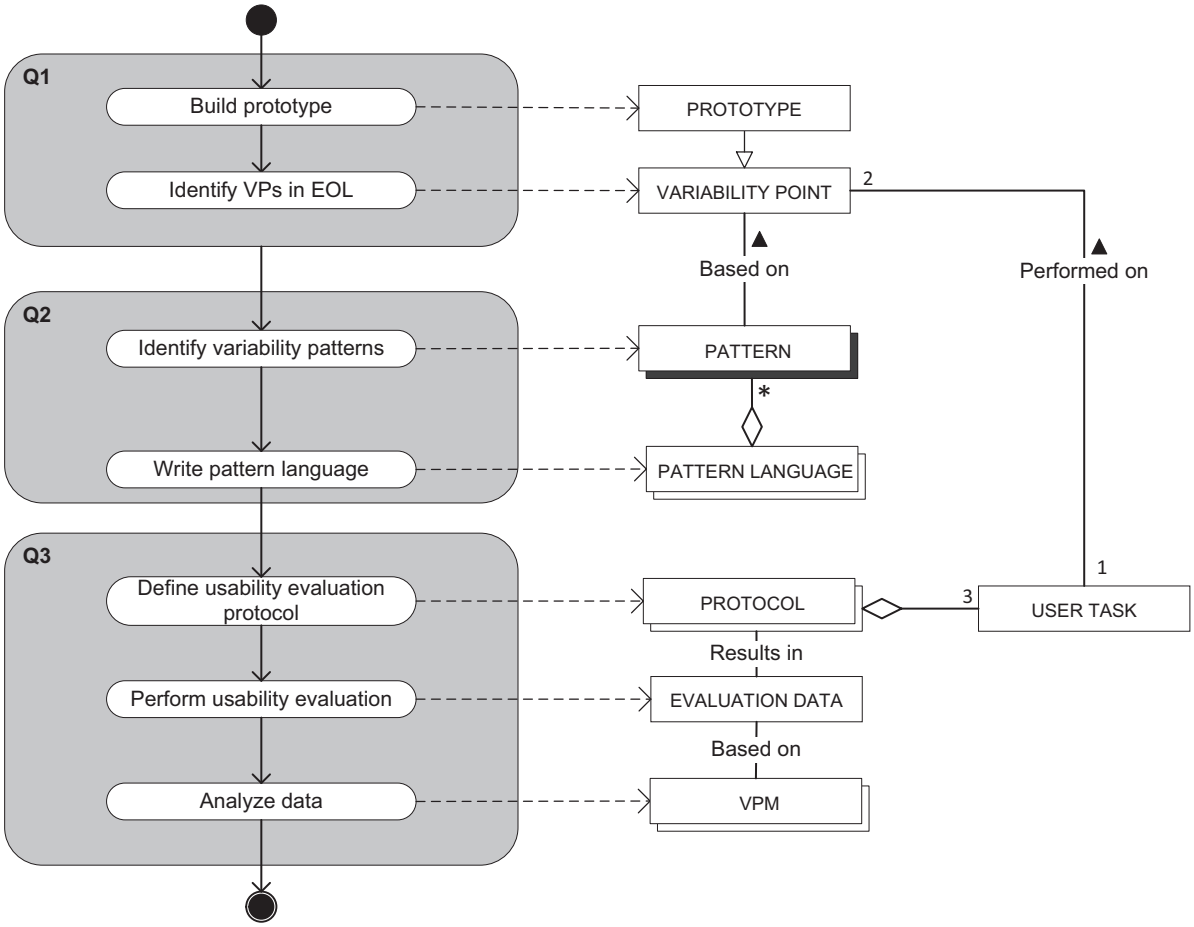

Figure 7 – The (simplified) research process captured within a PDD

Concept	Description
PROTOTYPE	This VP bridges the gap between the tenant-data trapped into EOL and the third party reporting tool Google Drive Spreadsheets. Needless to say, this PROTOTYPE is an example of a VARIABILITY POINT as well.
VARIABILITY POINT	A VARIABILITY POINT (VP) is a point in software facilitating a delayed design decision (Van Gurp et al., 2001). This research is focused on variability points facilitating tenant-dependent reporting.
PATTERN	A design pattern is a general reusable solution to a frequently occurring problem in a certain context (Gamma et al., 1994). The PATTERNs written in this study are design patterns specialized on solving problems related to the implementation of variability. For this reason, the PATTERNs identified during this research are variability patterns.
PATTERN LANGUAGE	A PATTERN LANGUAGE is a collection of interconnected PATTERNS that together describe one or more solutions to large or complex design problems (Alexander, Ishikawa, & Silverstein, 1977). The PATTERN LANGUAGE resulting from this research is focused on solving problems related to the implementation of tenant-dependent

	reporting within multi-tenant environments.
PROTOCOL	The usability evaluation PROTOCOL describes how the usability evaluation is designed, conducted, and how the results should be analyzed.
USER TASK	The PROTOCOL also defines three USER TASKS which participants of the usability evaluation have to complete using selected VARIABILITY POINTs. As can be seen in the PDD, each participant performs each USER TASK twice. One on a VP providing a relatively high flexibility and one on a VP providing a relatively low flexibility.
EVALUATION DATA	Besides the qualitative data – which is inherent to the thinking-aloud protocol – that resulted from the evaluation sessions, the EVALUATION DATA also comprises quantitative usability measures such as the time participants required to perform a task.
ANALYSIS	An ANALYSIS on the EVALUATION DATA is performed answering whether there exists a negative relationship between software variability (i.e. flexibility) and usability.

Table 3 – The concepts describing the research deliverables

3 Theoretical background

The major topics involved in this research are multi-tenancy, software variability, software patterns, usability and business intelligence. Multi-tenancy provides the context in which the business needs addressed by this research occur. Tenant-dependent reporting is an example of a software variability problem which has the potential to be solved by the implementation of specialized software patterns – i.e. variability patterns. Reporting is an important subject within the business intelligence (BI) area, thus in that sense this research is focused on tenant-dependent BI in multi-tenant environments. In the next sub sections, theoretical background on the four main topics is provided and for the sake of scope and clarity, important concepts are defined. This chapter concludes with an overview of the related work that combines the topics of variability and multi-tenancy.

3.1. Multi-tenancy

3.1.1. Multi-tenancy versus Software as a Service

In literature the terms SaaS and multi-tenancy are often used interchangeably and although both concepts are closely related to each other, we aim to provide clarity by making a clear distinction between the two. The Software as a Service (SaaS) deployment method is – at the expense of the traditional on-premise model – increasingly adopted by both software suppliers and users (Kaplan, 2005). On-premise software refers to software produced by a software vendor that is deployed at the customer's site whereas SaaS implies that the software vendor is the responsible party for hosting the software. A simplistic but clear definition of SaaS is (Chong & Carraro, 2006):

> *Software as a Service is Software deployed as a hosted service and accessed over the Internet.*

This shift of costs and responsibility (e.g. acquisition and management of the IT infrastructure and application maintenance) from the customer to the software vendor has changed the way in which customers are billed (Chong & Carraro, 2006; Lehmann & Buxmann, 2009). With on-premise software, customers typically pay a relatively high fee up front, whereas the recurring costs (e.g. hosting and maintenance costs) for vendors adopting the SaaS business model force them to send recurring invoices to their customers. Besides the *shift* of costs (from the customer to the vendor), the SaaS business model should cause a *decrease* of costs. The amount of money spend on the acquisition and maintenance of IT infrastructure and the maintenance of the application and databases running on top of it can now be shared among all customers using an application.

Multi-tenancy can be seen as an architecture for SaaS applications aimed at supporting the sharing of a single running-instance and database amongst multiple customers (Zhang et al., 2010). In other words, a software product is considered to be multi-tenant when the needs of multiple customer organizations are satisfied with a single stack of hard- and software resources (e.g. server, application instance, database) (Bezemer et al., 2010; Weissman & Bobrowski, 2009). In this context, a customer organization

is called a tenant whereas the actual users – employees of a tenant – can be denoted as tenant-users (Jansen et al., 2010). The major benefit of multi-tenancy is the improved server utilization rate that is achieved by multiple tenants sharing the same server (Chang Jie Guo, Sun, Huang, Wang, & Gao, 2007). For this reason, multi-tenant applications are especially interesting for small and medium sized customers (SME) due to the typical low hardware utilization they would have with on-premise software. The IT budgets are generally limited and there is no need for a dedicated server. When multiple tenants share the same IT infrastructure, the utilization rate can increase drastically (Warfield, 2007). Another means to share hardware resources is the use of a virtual server, however, due to the high memory requirements of virtual servers, the maximum number of tenants that can host their software on one physical server is much lower than the number that can be achieved by applying multi-tenancy (Li, Liu, Li, & Chen, 2008). To summarize, SaaS represents the business model in which the software vendor offers an application and the underlying IT infrastructure as a service (Kaplan, 2007), whereas multi-tenancy is the architectural pattern that should be implemented in order to exploit the resulting economies of scale to the full extend (Kwok et al., 2008).
SaaS products can be categorized into *line of business services* and *consumer-oriented services* (Chong & Carraro, 2006). Line of business services are often aimed at supporting business processes and thus offered to professional organizations. SalesForce and Exact Online are examples of the line of business services. Consumer-oriented services are – often free of charge – offered to consumers. These services are often monetized using advertisements, with Google's Gmail as a famous example. Obviously, this research concerns the line of business services category of SaaS, as this category is the one with tenants involved.

3.1.2. Multi-tenant versus Multi-instance

Another example of how SaaS can be implemented is the multi-instance architecture. With this approach, a code instance is hosted separately for each tenant. Applying this approach avoids the complexities that are inherent to the multi-tenant model. However, the maintenance burden increases with each new tenant as more and more running-instances have to be kept up to date (Chang Jie Guo et al., 2007).

3.1.3. Multi-tenant versus Multi-user

The main difference between multi-tenant- and multi-user applications lies in the configurability of the software. A multi-user application has to deal with multiple users whereas a multi-tenant application deals with multiple users employed at multiple tenants (Bezemer & Zaidman, 2010). Therefore, multi-tenant software does not only need to deal with customization on user level (e.g. changing the look and feel of the GUI) – as is the case with multi-user applications – but also on tenant level (e.g. customizing the application workflow to make it fit with the business processes of the tenant). To summarize, multi-tenant applications are multi-user, but a multi-user application is not per definition multi-tenant. The challenges from a technical perspective are not that different. Scalability, data separation, security and even variability are amongst the technical concerns of both application types. However, we can distinguish between variability implemented at the granularity of the tenant or the end-user.

3.1.4. SaaS versus PaaS

Although Software as a Service and Platform as a Service are two different concepts, they are strongly related to each other. A PaaS is an application centric approach that abstracts not only the concepts of IT infrastructure – often marketed as Infrastructure as a Service (a.k.a. IaaS) – but also concerns related to multi-tenancy, performance (e.g. load-balancing) scalability, availability, security and even back-ups (Weissman & Bobrowski, 2009). In other words, a PaaS facilitates the rapid development of applications as developers are merely concerned with the development of true functionality (aspects of an application that are visible to customers). A PaaS that perfectly illustrates the close relation between SaaS and PaaS is the Force.com (owned by Salesforce.com). This PaaS provides all the tools for developers to develop a SaaS application on top of itself. In fact, the Salesforce.com CRM system is built on top of this PaaS. When a SaaS is developed on a PaaS, it generally has a high degree of software variability as the tools provided by the PaaS can be used to customize and extend the existing SaaS. In the case of the Force.com, applications that are built on top of it are automatically multi-tenant aware. To summarize, a multi-tenant SaaS can be built on top of a multi-tenant PaaS, and in such a case the development tools of the PaaS generally provide a high degree of tenant-based variability for the SaaS offerings running on top of it.

3.1.5. Variability and Multi-tenancy

Obviously, SaaS providers want to obtain as many paying customers as possible. For this reason, a successful SaaS product is one that is designed to satisfy the divergent requirements of customers within their target segment. C. J. Guo et al. (2011) introduced a competency model that is aimed to guide SaaS providers with designing the configuration and customization aspects of their services. The importance of customization in multi-tenant software comes to light in the definition of the 'multi-tenant application' coined by (Bezemer & Zaidman, 2010):

> *A multi-tenant application lets customers (tenants)* ***share*** *the same hardware resources, by offering them one shared application and database instance, while allowing them to* ***configure*** *the application to fit their needs as if it runs on a dedicated environment.*

This definition is twofold as it describes the sharing aspects of multi-tenancy but also emphasizes the need to allow tenants to configure the application to fit their needs. In a single-tenant environment the software is often customized because every customer has its own special requirements and customization is relatively easy to achieve since they are the only tenant utilizing a running instance. A multi-tenant SaaS application has both financial and flexibility advantages, however, for many potential customers these advantages would not outweigh the disadvantage of the lack of customization. For this reason, variability in multi-tenant software is essential to satisfy tenant requirements (Mietzner et al., 2009).

3.1.6. Multi-tenancy Levels

Based on implementation complexity the following multi-tenancy levels are defined by Kabbedijk and Jansen (2011):

- *Data model multi-tenancy:* tenants share the same database;
- *Application multi-tenancy:* besides the database tenants also share a single code instance of the application;
- *Full multi-tenancy:* Like application multi-tenancy, tenants are sharing a single database and code instance but now the application has a certain level of variability which enables tenants to have their own variant of the application.

Multi-tenancy is all about sharing resources amongst multiple tenants and when this is done only at data model level, we define this as data model multi-tenancy. Multi-tenant databases can be implemented in various ways including the shared machine, shared table, and shared process approach (Jacobs & Aulbach, 2007). The second level describes standard multi-tenant software whereas the last level refers to true multi-tenant environments as it implements a certain level of variability that enables tenants to adapt the software to fit their business context.
A slightly different distinction of multi-tenancy levels is made in the Saas Maturity model presented by Chong and Carraro (2006). Ultimately, a SaaS application is scalable (e.g. maximized concurrency, sharing pooled resources, and partitioning large databases), multi-tenant efficient and highly configurable, however, for some SaaS provider it can be more cost efficient to aim for a lower SaaS maturity level (Chong & Carraro, 2006). In the software as a service maturity model presented four maturity stages are distinguished. The first maturity level is quite similar to the application service provider (ASP) approach. Each tenant has its own customized code instance hosted by the SaaS provider which enables easy tenant specific customizations since the code instances are fully independent of each other. Unfortunately, the simplicity of the customization goes at the expense of an increased complexity when the software product needs to be updated. For these reasons, the maintenance costs are relatively high, resource sharing is rather limited, but tenant specific customizations are easily implemented. At the second SaaS maturity level there is still one hosted code instance per tenant, however, they're now identical to each other which simplifies the updating of the code instances to a newer product version. A consequence is that tenant specific customization cannot be done by adding custom code to the code base, thus at this maturity level, a configuration meta-data approach is applied to enable tenants to adapt the software to their context. Since this approach is configurable, it satisfies one of the three attributes that an ultimate SaaS application should poses. To summarize, the approach of the second level limits customization possibilities but incurs less maintenance costs. Investments in hardware resources are similar with the first maturity level. The third level differs from the second in that all tenants are sharing the same code instance. This approach is both configurable and multi-tenant efficient which facilitates SaaS vendors to profit from the economies of scale as not only the hardware resources but also the code instance is shared. The weakness of this approach is the limited scalability as the only ways are moving to more powerful hardware or applying the tenant-placement approach in which the load on database level is balanced by dividing tenants over multiple databases. SaaS applications with level four maturity are not only configurable and multi-tenant efficient, but also fully

scalable as a tenant load balancer is used to divide tenants over a pool of code instances. The reason that we use the term load-balancing on code-instance level and tenant placement on database level is because load-balancing typically implies statelessness (with stateless sessions, every request can be handled by a different webserver) whereas tenant placement refers to a stateful (persistent) distribution of tenants over multiple databases.
This research is focused on applications that are on the full multi-tenancy level in terms of Kabbedijk and Jansen (2011), or applications on both the third and fourth level in terms of the SaaS maturity model by Chong and Carraro (2006).

3.2. Tenant-based Variability

The notion of variability in the software domain is introduced in in software product lines where customer specific requirements are satisfied by generating different products based on one comprehensive product core containing all customer requirements (Pohl, Bockle, & Van Der Linden, 2005; van Gurp et al., 2001). Variability realization techniques describe how variation points – i.e. points in software facilitating a delayed design decision (Van Gurp et al., 2001) – can be implemented (Svahnberg et al., 2005). These techniques can have different moments in the software lifecycle at which the choice to use a specific variant has to be made – i.e. different binding times (Jansen et al., 2010). With a multi-tenant SaaS product, all customers use the same instance of the software and as a result, it is detrimental to redeploy the product every time changes are made. On-premise products resulting from a software product line are compiled before shipping. For this reason, as opposed to software product lines, multi-tenant environments only profit from run-time bound variability realization techniques.
Within multi-tenant SaaS, variability can originate from tenant specific requirements (tenant-oriented variability) or segment specific requirements – i.e. segment variability (Kabbedijk & Jansen, 2011). A segment is a group of tenants that have specific requirements in common. Examples of segment variability are different languages, tax rules or currencies for segments based on geographical location, or different workflows for segments based on organization size. Examples of tenant-oriented variability is when the software enables tenants to extent the data model with context specific data objects, to add or delete steps in the application workflow or when tenants can choose between different third party reporting tools to generate BI based on the data from the multi-tenant environment. This research concerns run-time tenant-oriented variability since we focus on tenant-dependent reporting in multi-tenant SaaS.
Variability, customization and configuration are often used to describe the very same concept. For instance, during the search for patterns that can be implemented to enable tenant-dependent reporting, we have found customization realization techniques (Jansen et al., 2010) and variability realization techniques (Kabbedijk & Jansen, 2011; Svahnberg et al., 2005), two different names used for the same concept. However, these terms are not redundant and all have a unique meaning. For software variability, we adopt the definition of (van Gurp et al., 2001):

> *Software **variability** is the ability of a software system or artifact to be changed, **customized** or **configured** for use in a **particular context**.*

This definition states that customization and configuration are both ways to implement software variability, which ensures that tenants can fit the software – to certain extent – into their particular context. The difference between configuration and customization can be explained by the level of complexity involved. The simpler of the two is configuration, which provides tailoring within a pre-defined scope. Adding business rules, changing workflows or even a report builder within a bookkeeping system are examples of configuration. Customization involves changes to the source code of the application and is therefore a method that adds more variability but also more complexity and costs (Sun, Zhang, Guo, Sun, & Su, 2008). Compared with configuration, customization results in a higher variability because there is no pre-defined limit involved. The complexity is increased as for every tenant, SaaS providers need to maintain one or more pieces of customized code. Updating to a newer version of a multi-tenant application, while preserving the customized code and backward compatibility, requires a well-designed architecture. Customization is more costly than configuration because customizing source code requires higher educated/skilled employees and this form of tailoring usually takes a lot more time than the configuration of an application. Other reasons for higher costs include the fact that software maintenance will require more resources and SaaS providers can miss out on customers that not accept the higher costs and/or complexity involved (Rohleder, Davis, & Günther, 2005). Based on these arguments, SaaS providers should tenantize (working towards the tailoring requirements of tenants) their applications by configuration instead of customization wherever possible.
The need for variability in multi-tenant environments originates from the divergence in the requirements from tenants. The most examples of variability in multi-tenant applications are based on functional requirements from tenants. Note however, that tenants can also have non-functional (or quality) requirements which are for instance security or availability related (Mietzner et al., 2009; Tsai, Shao, & Li, 2010). Within Exact Online (the multi-tenant application of the case company) for example, some tenants have a database for themselves, while clients that do not have this specific quality requirement share a database with multiple other clients.

3.2.1. Visibility

In the software product line engineering discipline, a distinction between external and internal variability is made (Andreas Metzger & Pohl, 2007; Pohl et al., 2005). External variability is the visibility that is visible to the tenants whereas internal variability is only visible to the developers of the software product line. In the multi-tenant SaaS context, customer-driven variability and realization driven variability are distinguished (Mietzner et al., 2009). The former is similar to external variability, whereas the latter is equivalent to internal variability. This distinction is important in the modeling of variability in multi-tenant SaaS environments, which support SaaS providers in their variability decision making (Mietzner et al., 2009; Sinnema & Deelstra, 2007).

3.2.2. Variability Levels

In literature, multiple levels of variability are distinguished based on the level of flexibility they provide to tenants. One of the more elaborate distinctions is made by Sun, Zhang, Guo, Sun, and Su (2008) who define 5 SaaS configuration/customization maturity levels (hereafter referred to as variability levels) based on the level of variance requirements supported by the application. By increasing the maturity level of the SaaS application, greater variance of requirements is supported and as a result, a larger and broader customer base can be served. From completely standardized to fully 'tenantized' SaaS, the following variability levels have been defined by Sun et al. (2008):

- *Entry:* highly standardized offering without any configuration and customization support*;*
- *Aware:* relatively standardized offering with pre-defined variance points*;*
- *Capable:* relatively standardized offering with user defined configuration*;*
- *Mature:* base offering with programmable environment to enable user preferred customization;
- *World Class:* offer a platform supported by programming model and tools to enable extremely strong customization or even new application development.

Variability levels can also be defined by the various architectural layers at which variability can be implemented. The Model-View-Controller (MVC) model separates architectural layers based on the type of logic involved (Krasner & Pope, 1988). The model layer contains the business logic of the application, i.e. the data business rules implemented by the data model containing business objects with their attributes, methods, and mutual relationships. The controller layer contains the application logic which implements the application workflow whereas the view layer contains the presentation logic which for instance determines how data from the model layer is presented to the end-user. Based on the MVC architecture we can distinguish between the following variability levels in tenant-oriented variability (Kabbedijk & Jansen, 2011):

- *Low*: variability at this level only impacts changes in the view layer;
- *Medium*: tenants are able to customize the logic in the controller or model layer;
- *High*: variability at this level impacts multiple tiers at the same time.

Examples of low level variability include the possibility for tenants to change colors in the user-interface or the ability to sort or filter records in some overview. Medium variability includes the ability to create new data objects or add attributes to existing objects. An example of high variability is when tenants are able to run their own program code.

3.2.3. Variability Classifications

There are five variability realization techniques (VRTs) defined by Svahnberg et al. (2005) that are run-time bound which makes them applicable for multi-tenant environments. Based on these VRTs, Jansen et al. (2010) defined five customization realization techniques (CRTs) which enrich the multi-tenant customization vocabulary by considering the scale on which the variability is applied. The five CRTs are divided in two major customization types, namely model-view-controller customization and, system

customization. Obviously, three of the CRTs belong to the MVC customization type, one for each layer of the MVC architecture (Krasner & Pope, 1988). The remaining two CRTs – connector and component– belong to the system customization type. The MVC CRTs are different from the variability levels distinguished by Kabbedijk and Jansen (2011), although both the variability levels and CRTs are defined based on the MVC layer in which certain variability is implemented. A CRT is defined for each layer of the MVC architecture while the variability levels of Kabbedijk and Jansen (2011) use the MVC layers in which variability is realized together with the amount of MVC layers that are affected by one variability implementation to indicate the level (e.g. extensiveness, complexity, and provided flexibility) of variability. In terms of CRTs, a view change (e.g. tenants can change colors of the UI) is an example of low variability, whereas a model or controller change (both CRTs) is an example of medium variability. When a variability implementation involves model, view, and controller changes we can classify this as high variability. When a multi-tenant application has the ability to connect with different external applications providing similar functionality, this is called system connector customization. An example of the system connector CRT is a multi-tenant bookkeeping system when reports can be generated either in Google Drive Spreadsheets or Microsoft Office 365, depending on the tenant's preferences. The system component CRT is similar to the connector CRT with the main difference that a component change refers to the situation in which the tenant can choose between multiple components providing similar functionality whereas a connector change refers to multiple connections to external systems. An example of system component change is when tenants can either share a database with other tenants or have a database for their own, depending on their security requirements.

A quite different classification on software variability realization techniques can be made based on the location at which the customizations are hosted and executed: desktop integration, user-interface customization, and back-end customization. Desktop integration refers to integrations with SaaS product, executed on the client-side (in contrast with the SaaS product itself, which runs server-side). User-interface customizations extend the functionality by consuming external services which are embedded as UI-widgets within the front-end of the SaaS product. This is similar to the way (HTML) iFrames are used to embed external content within a web page. However, the widgets might interact with the SaaS product, i.e. by exchanging data or affecting the application workflow. Back-end customizations – the most invasive of the three – are hosted and executed on the back-end of the SaaS product. This allows developers to extend or replace the business logic of the application.

3.2.4. Granularity

Generally, the term granularity is used to indicate the extent to which an entity can be subdivided into smaller entities, i.e. the amount of detail in a hierarchical structure. Variability in a multi-tenant environment can be implemented at different levels of granularity (Li, Shi, & Li, 2009; Tsai et al., 2010). Li et al. (2009) proposed a customization relationship model is presented which states that variability is generally applied on objects within four technical levels, that is to say the data-, service-, process-, and user-interface level. In addition, the model also describes four variability granularity levels. Before we elaborate on these granularity levels, we briefly describe the objects residing within the four technical levels.

Variability on the data-level is done by enabling tenants to instantiate data objects – representing business entities – and data fields – describing attributes of business entities. A service implements a specific function and is either atomic or a composition of atomic services. An atomic service has the responsibility to perform a small fundamental task whereas a composite service is used to perform complex tasks by invoking atomic services required for completing the tasks. Services are not only provided by the SaaS provider but can also be delivered by external parties (parties other than the SaaS provider or the tenant). Li et al. (2009) defined two relationships among services, namely invoke and confinement. The invoke relationship refers to the composite relationship between atomic and composite services whereas confinement refers to rules regarding the sequence and mutual exclusivity of services. Some services should be performed in a prescribed order, e.g. the service generating a graphical representation of some data can only be executed after the query service has been performed. Services that are not meant to be executed both in the same process, e.g. the data is represented either by the graph generating service or by the service generating a data grid. The process level of a multi-tenant application involves the organization between services, data and process participants (i.e. actors). Data is used and generated by services and it is input for the conditions that shape the process (process step transformation). On the user-interface level reside objects representing pages, menus and composites where the page is built up from composites, e.g. forms or lists, and the menu is used to navigate to pages.
The four granularity levels defined by Li et al. (2009) are listed and briefly explained in the table below.

Table 4 – Variability granularity levels

Granularity Level	Description	Flexibility	Difficulty
Parameter	The objects used in the application are fixed although the existing objects can be configured within pre-defined scope.	Low	Low
Object	The application allows tenants to create customized objects via the GUI, e.g. using wizards or templates.	Middle	Middle
Cooperation	At this level tenants are able to combine the standard and customized objects from the various technical levels to orchestrate processes that complete certain tasks.	High	High
Coding	The multi-tenant environment contains a platform which enables tenants to add or customize functionalities (e.g. UI components or business rules) by adding source code.	Highest	Highest

A nice example of parameter granularity is when a tenant is able to change the colors of the GUI or when tenants can choose between multiple pre-defined workflows. An example of object variability is when tenants are able to extend the standard business model with entities (including attributes and relationships with other entities) that are specific for their business domain. Multi-tenant applications in which these customized entities are combined with available services (both from the SaaS provider and third parties) to compose new processes able to complete tasks that are specific for the business of one tenant, has implemented variability on cooperation level. APEX, the strongly typed programming

language executed on the Force.com platform, is probably the most famous example of variability on the coding granularity level (Weissman & Bobrowski, 2009).
The ontology-based customization framework proposed by Tsai et al. (2010), requires multi-tenant applications to have a multi-layered architecture. These layers, the objects they contain, and the relations inside and between layers, are identical to the technical levels described by Li et al. (2009) although they are called GUI-, business process-, service-, and data layers. To enable easy customization, domain ontologies are used to create templates that can be customized by tenants by selecting a subset of the objects from a specific level. To provide an indication of complexity, a distinction between heavy and lightweight variability is made. From light to heavy, four customization granularity levels are defined; parameter-, entity-, composition-, and implementation customizable. Just like the layers, the granularity levels are very similar to those described by Li et al. (2009). Besides the names, the only differences is that variability at entity granularity is specific to objects in the data layer (business entities) whereas the object granularity is less specific defined (see the table above).
An interesting conclusion that we can draw is that the granularity of a variability pattern can serve as a first indication of its relative flexibility and complexity. In addition, a tradeoff is observed as variability patterns that facilitate a higher flexibility imply an increasing complexity of both the implementation by the SaaS vendor and the usage by the tenant (Li et al., 2009; Tsai et al., 2010).

3.2.5. Tenant Perspectives

Sun et al. (2008) provided an overview of the tenant perspectives on configuration and customization requirements is provided including the seven main categories; organization and structure, workflow/process, data processing, data, user interface, report, and business rules. These perspectives are introduced to facilitate more accurate analysis and evaluation of variability maturity of a multi-tenant environment.
The report configuration and customization perspective – the one that is relevant for this research – consists of three sub items; add/change dataset, add/change query rules, and add/change report style (e.g. chart, table graph). Both report and user interface configuration and customization are defined as top level perspectives, but a change impact relationship from the former to the latter suggests report variability to be a subset of user interface variability. In addition, the sub items of the report perspective can be grouped under other main categories, for instance, the add/change report style sub item can also be seen as a more concrete instantiation of the user interface perspective.

3.3. Variability Patterns

Software patterns are a suitable means for the communication and documentation of solutions for the implementation of variability within multi-tenant environments (Kabbedijk & Jansen, 2011, 2012). For this study, we adopt the definition of design patterns provided by the Gang of Four (Gamma, Helm, Johnson, & Vlissides, 1994) to define the concept of software pattern:

A design pattern is a general reusable solution to a frequently occurring problem in a certain context.

The concept of patterns originates from the architecture of buildings and towns (Alexander et al., 1977) which inspired the Gang of Four (GoF) to introduce design patterns in software engineering. The definition of design patterns provided by the GoF is quite general and does not mention a specific problem domain, though design patterns address problems in software engineering on the level of the object-oriented paradigm. While this research does not exclude patterns on object-oriented design level, it certainly includes patterns describing software engineering solutions on a higher level – i.e. architectural patterns (Buschmann et al., 1996). Besides design patterns and architectural patterns, software patterns refer to various kinds of software related patterns including data-model patterns, user-interface patterns, distributed-programming patterns, and analysis patterns (Fowler, 1997). For this reason, we use the more general term 'software pattern' to refer to a general reusable solution to a frequently occurring software engineering problem in a certain context.

The definition provided by the GoF qualifies solutions as patterns only when they solve a recurring problem. An additional qualifier is that these solutions should have a certain degree of effectiveness in solving the frequently occurring problem (Kabbedijk & Jansen, 2012). When a solution to a specific problem has the required recurrence to be called a pattern but in addition introduces major adverse consequences, it can be considered an anti-pattern (Brown, Malveau, & Mowbray, 1998). In many cases multiple software patterns can be applied to solve a problem, though some will be more effective than others. Unfortunately, the effectiveness of a solution is hard to define and depends on the problem context. Imagine for instance a software pattern that significantly improves the reusability of components within an application but at the expense of its performance. This will be a serious liability for a bookkeeping system with over 25k tenants. However, the design pattern can be a perfect solution for a specialized application aimed at serving 7 tenants.

As it is hard to develop quantitative measures of the effectiveness of a software pattern, their effectiveness cannot be validated. However, when a fixed format is used to thoroughly document the consequences of implementing a pattern, the drivers of the problem the pattern solves, and the relations between drivers and consequences, the effectiveness of software patterns can be evaluated. The most used pattern documentation forms include the *Alexandrian* style (the oldest) originating from Alexander's pattern book (Alexander et al., 1977), the GoF form (Gamma et al., 1994), and the POSA form – originating from the pattern oriented software architecture book (Buschmann et al., 1996). Another common used pattern structure is the Coplien form named after Jim Coplien who frequently applied this style in his books (Coplien & Schmidt, 1995). Within the Coplien style, the pattern is described by elaborating the problem, context, forces and solution.

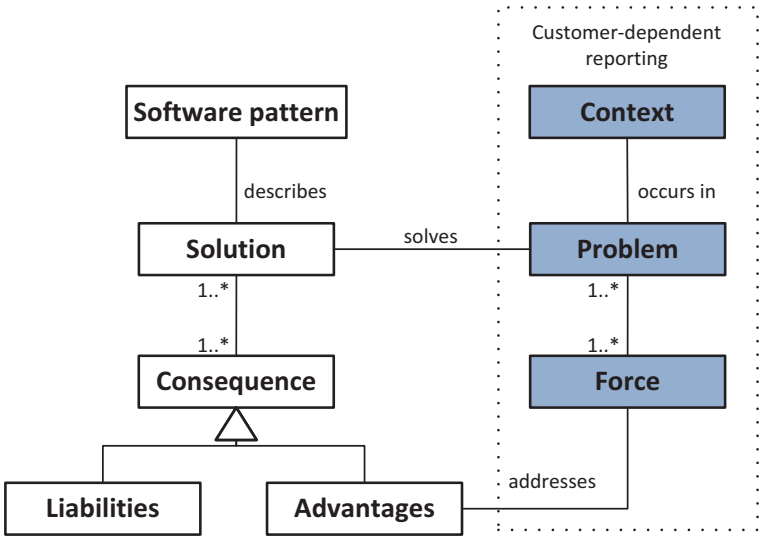

Figure 8 – The role of variability patterns

Kabbedijk and Jansen (2012) – who suggested that software patterns play an important role in the implementation of variability in multi-tenant environments – propose a similar approach. A software pattern describes a solution which solves a problem driven by one or more forces occurring in a specific context. As shown in the conceptual model of figure 8, the patterns elaborated in this research solve problems in a context related to tenant-dependent reporting. The implementation of the pattern results in a solution incurring one or more consequences which are either advantages or liabilities. When the forces that form the problem are thoroughly identified and explicitly listed, the consequences of implementing the pattern can be related with these forces. The better the forces are addressed by the consequences (advantages), the higher the effectiveness of the pattern (Kabbedijk & Jansen, 2012). As consequences are not always advantageous, we also need to take the liabilities of a pattern into account in order to truly judge on the effectiveness of a pattern. When liabilities and possible mitigations are elaborated, and the advantages are related to thoroughly described forces, we can judge on a pattern's effectiveness.

3.4. Business Intelligence

In statistics, a probability distribution has a 'long tail' when a large part of the population falls far from the mean. This concept is used to illustrate a retailing strategy (figure 9) in which a large number of unique unpopular items are sold in small quantities – e.g. as succesfully applied by Amazon.com (Anderson, 2004). The economies of scale introduced by multi-tenancy allows the low-end market to access specialized line-of-business solutions previously merely accessible for large companies able to pay for the vendor services required to tailor software to their specific needs, and purchasing and managing the IT infrastructure required to host the software. Since software variability is one of the key attributes – besides scalability and tenant-isolation – required to successful exploit a multi-tenant software products, this should be implemented to meet a broader set of customer requirements and thus sell the product in larger quantities – i.e. catching the long tail.

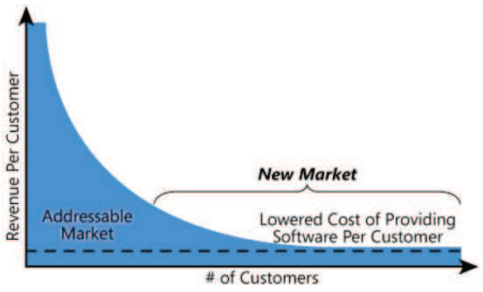

Figure 9 – Catching the long tail by Chong & Carraro (2006)

Business Intelligence (BI) is an example of an ability that used to be out of reach for the low-end market due to its costs and complexity. However, when SaaS providers combine the multi-tenancy and software variability concepts, affordable tenant-dependent reporting functionality (both as standalone products or as part of online business solutions) can enable simplified BI for small and medium sized enterprises (SME).

In 1958 the term business intelligence was introduced, referring to *"the ability to apprehend the interrelationships of presented facts in such a way as to guide action towards a desired goal"* (Luhn, 1958) whereas Howard Dresner popularized BI in the early 1990s as the umbrella term as it is used nowadays, referring to *"a set of concepts and methods to improve business decision making by using fact-based support systems"* (Power, 2007). Using this broad and comprehensive definition, BI not only includes obvious functions like reporting, dashboards and analytics, but also functions belonging to the data (and information) management segment such as data -warehousing, -quality, and -integration

(Evelson, 2008). For this reason, BI is sometimes split up into data preparation and data usage in order to distinguish between the lower and the upper layers of the BI stack (Evelson, 2010).

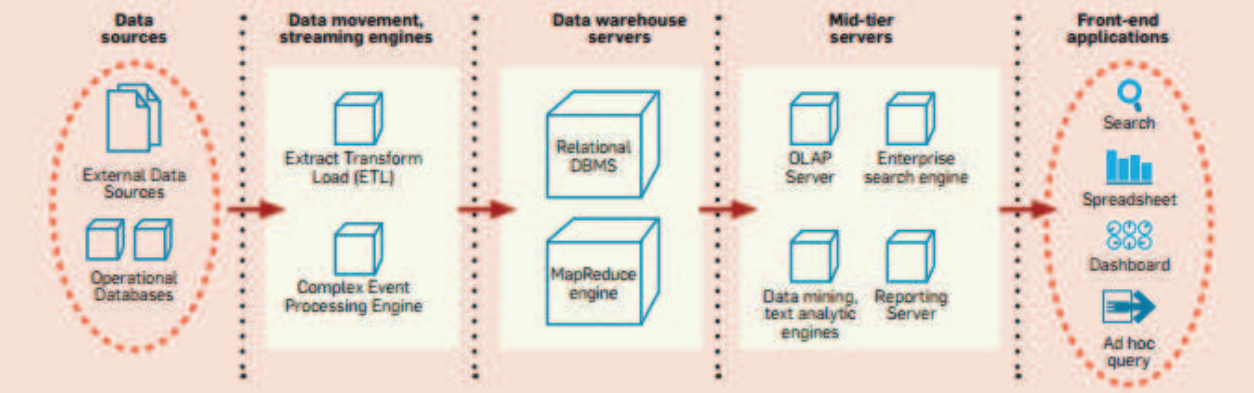

Figure 10 – A typical business intelligence architecture (Chaudhuri et al, 2011)

A typical (simplified) business intelligence stack having the data preparation layers at the left and the data usage layers on the right is depicted in figure 10. The focus of this research – i.e. the proposed tenant-dependent reporting solutions – is on the data usage part of BI. To be more specific, the core BI features – reporting (the focus of this research) and analytics – are referred to as the presentation layer of BI (Evelson, 2008).

3.4.1. Tenant-dependent Reporting

When software variability is implemented within the functional area of reporting, this can be referred to as tenant-dependent reporting (TDR). The reason to focus on variability within one functional area is to enhance the comparability and validity of research results. By selecting a functional area that became available for the low-end market due to the economies of scale introduced by multi-tenancy architectures (recall the long tail strategy), this research became even more relevant.

In addition, the TDR patterns proposed in this research address the needs of the larger enterprises as well. Kobielus, Karel, Evelson, and Coit (2009) elaborate on the need for Do-It-Yourself Business Intelligence. Self-service BI can be utilized to cut BI costs and enhance BI decision support (Kobielus et al., 2009). Even in the most user-friendly tools, the design of reports is so complex that knowledge workers (the BI users) are generally unable to create these themselves. For instance, in the *"Forrester August 2008 Global BI And Data Management Online Survey"* 82 IT decision makers were asked who develops their BI artifacts (e.g. reports) and 70% responded that these are created by the IT department (Kobielus et al., 2009). As a result, a BI service backlog arises which needs to be processed by BI developers. Due to the current economic turndown IT budgets decrease while the need for BI, and thus the size of the accompanying BI service backlogs, did not change. As with self-service BI, the goal of tenant-dependent reporting is to empower users to obtain exactly the knowledge they need without the help of BI specialists. When this goal is reached, BI costs will decrease as expenditures on BI specialists become (to a certain extent, depending on the type of BI demands) obsolete. In addition, knowledge workers do not have to wait until their BI requests are processed which enables them to respond faster

to business events. This research does not only propose solutions for TDR, but also addresses their usability in relation with the amount of provided flexibility.

3.5. Usability

As depicted in figure 11, we expect a lower perceived usability (dependent variable) with variability points providing high flexibility (independent variable) then with variability points providing low flexibility. Besides the DV and the IV, two extraneous variables (EV) are shown in the conceptual model. The experience that a user has with similar software functionality might influence his perceived usability. Highly experienced users are likely to deal better with the complexities inherent to high flexibility than users that never performed a similar task before. Another factor influencing perceived usability might be the graphical user interface (GUI) of the application. Highly complex functionality can get easier to use when there is a very good intuitive GUI on top of it while even very simple functionality can become hard to understand when a very bad GUI is involved. There are various ways to counteract on these rival explanations – as referred to by Yin, (2008) – forming a threat to internal validity. Since the GUI and the experience level are attributes of the variability point and tenant-user respectively, we can lower the risk of distorted results by selecting variability points with similar GUIs and tenant-users with similar experience for customer-evaluations. Details on this and other measures related to the data collection protocol are elaborated in chapter 6. However, we can also reduce risks by creating a sound and unambiguous working definition of the usability concept. By doing this, we address the construct validity – i.e. the extent to which we actually measured what we meant to measure. The remainder of this section is spent on the operationalization of the usability concept.

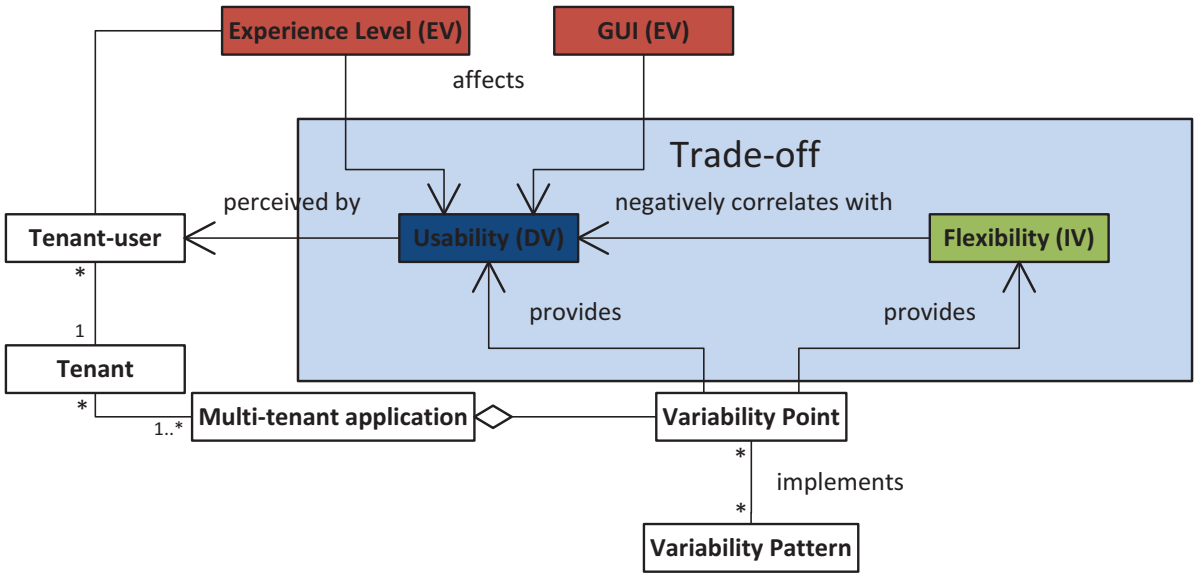

Figure 11 – Conceptual model of the usability versus flexibility trade-off

3.5.1. Operationalization

As with most concepts, there are multiple authors and standards that have attempted to define the concept of usability and there is a lack of a comprehensive method to measure and evaluate software products on this quality attribute (Seffah et al., 2006). Although this study is not addressing the question of what usability really is, we need a very clear definition of the usability that is truly inherent to the implementation of software variability. Two examples of high level definitions originating from well-known standards that largely correspond to our view of this type of usability are as follows:

- *"The extent to which a product can be used by specified user to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use"* (ISO 9241-11, 1998)
- *"The ease with which a user can learn to operate, prepare inputs for, and interpret outputs of a system or component"* (IEEE Std.610.12, 1990)

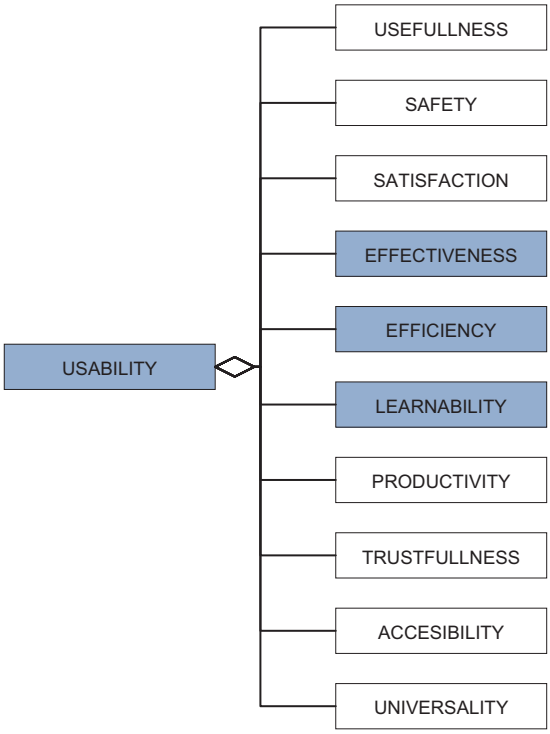

Figure 12 – Usability Factors

Both of the above definitions do certainly not exclude the usability characteristics that we deem to be inherent to variability points. However, they include various usability characteristics that are certainly not inherent to variability points. For this reason, we require a more narrow definition excluding all the irrelevant or inapplicable aspects of usability. Once we have a usability definition scoped to variability points, we need to operationalize this concept – i.e. decomposing a concept into multiple variables for which operational definitions or metrics can be defined.

We use the work of Seffah et al. (2006) – which describes a literature study across different usability standards and methods in order to create a more consistent and consolidated model that brings factors, criteria and metrics together – to select the factors, criteria, and metrics, that are relevant to the usability evaluation of this study. When looking at figure 12, one can see that the concept of usability can be split up into 10 usability factors of which three – learnability, effectiveness, and efficiency – are considered to be relevant for the usability evaluation of variability points. The relevance of these factors is explained in the remainder of this section. In addition, we elaborate on the importance of defining a context of use when measuring usability.

3.5.1.1. Context of Use

As is also suggested by the usability definition of the ISO 9241-11 standard, one can only make a proper judgment on the usability of a software system when the context of use is defined (Seffah et al., 2006). The 'context of use attributes' defined in ISO 9241-11 include the characteristics of the user, task (e.g.

frequency, duration, importance), technical environment (e.g. operating system, hardware capabilities and constraints), physical environment (e.g. noise level, privacy), organizational environment (e.g. structure of operational teams and the user's level of autonomy), and the social environment (e.g. degree of assistance available). For this research, the user characteristics are most applicable and those should thus be defined preparatory to the actual measurements. Relevant data on the user characteristics include task experience, application experience, system experience (e.g. knowledge of computer, OS and browser), and even psychological attributes like the cognitive style (e.g. motivation to use the system and analytic versus intuitive). The task experience relates to the knowledge of the domain, whereas the application experiences concerns the experience with similar applications.

To illustrate the importance of defining the context of use before measuring usability, consider a variability point that is empowering users to build their own queries which can be used to generate reports. This specific functionality might be too complex for a typical bookkeeper without any knowledge of data persistence. A business intelligence professional on the other hand, won't have any problems with effectively using this functionality.

When the goal of a usability measurement is to evaluate an application, the context of use should describe the target customer/user group the application was designed for. The goal of the usability measurement in this study is somewhat different as we want to gain knowledge about the relation between usability and flexibility. However, as the variability points of which the usability is evaluated are implemented within the multi-tenant environment of the case company, it makes sense to select participants for this evaluation similar to the users their system was designed for. Most customers of Exact Online are small companies in which the bookkeeping is done by the company owners themselves. Currently, 40% of the customer organizations are sole proprietorships, whereas 31% are companies with 2 to 4 employees. Obviously, entrepreneurs know about sales, revenue, balance sheets and income statements. However, the core task of the entrepreneurs is not bookkeeping, and for this reason, the product should empower users without a bookkeeping background to manage their accounting.

For the evaluation sessions for determining the usability provided by the identified variability points, participants with sufficient bookkeeping knowledge should be selected as results would get distorted when task performance degrades due to a lack of domain knowledge. Little system knowledge is required in order to use Exact Online. However, to avoid performance issues due to a lack of this knowledge, only participants with sufficient knowledge are selected for the evaluations. The familiarity of participants with the UI of the product is also a variable that should be controlled. On the one hand, experience with the UI will lower the risk of distorted results in the same way as system and domain knowledge can influence results related to task performance. On the other hand, selecting existing customers of Exact Online as participants might also be a threat to validity for two reasons. The variability points observed within Exact Online have a (slightly) different UI than the variability point implemented by the prototype. In addition, participants that are existing customers might be already familiar with one or more of the variability points tested, which could cause a difference in task performance amongst variability points. To solve all issues mentioned, participants should not be selected from existing customers but before the evaluation session, participants should be made familiar with the UI to make sure that a lack of familiarity will influence task performance. Measurements are most accurate when participants do not have experience with functionality similar to the variability points. In the ideal situation, we can control this variable by participant selection. An alternative way to

control experience with similar functionality is to include related questions in the survey filled in by the participants. When this variable actually influences task performance it can be detected during the data analysis.

3.5.1.2. Efficiency

Efficiency refers to the *"capability of a software product to enable users to expend appropriate amounts of resources in relation to the effectiveness achieved in a specified context of use"* (Seffah et al., 2006) – another sub variable of usability. Efficiency can be considered as one of the measurable criteria of learnability. When tasks are getting more complex, the efficiency might decrease. For this reason, we consider efficiency as one of the aspects of usability that is related to the concept we attempt to measure.

3.5.1.3. Effectiveness

Another sub variable of usability which is considered relevant to this research is effectiveness, which is defined as *"the capability of a software product to enable users to achieve specified tasks with accuracy and completeness"* (Seffah et al., 2006). Especially the completeness aspect of effectiveness can function as an indicator for task complexity.

3.5.1.4. Learnability

Learnability is by far the most relevant sub factor related to the usability of a variability point. When one expects usability to decrease with an increase of flexibility, in a sense, usability is used as a synonym of simplicity. In other words, the use case complexity increases when the flexibility provided by the variability point increases. The sub factor reflecting this complexity is learnability. A taxonomy of learnability definitions proposed throughout literature is presented by Grossman, Fitzmaurice, and Attar (2009). This taxonomy – depicted in figure 13 – is based on two main learnability categories, that is to say initial learnability and extended learnability. Extended learnability refers to the change in task performance over time whereas initial learnability concerns the initial performance in completing a specific use case with a software product. As depicted in figure 13, we have marked the parts of the taxonomy that constitute a learnability definition applicable to the concept we aim to measure during this research.

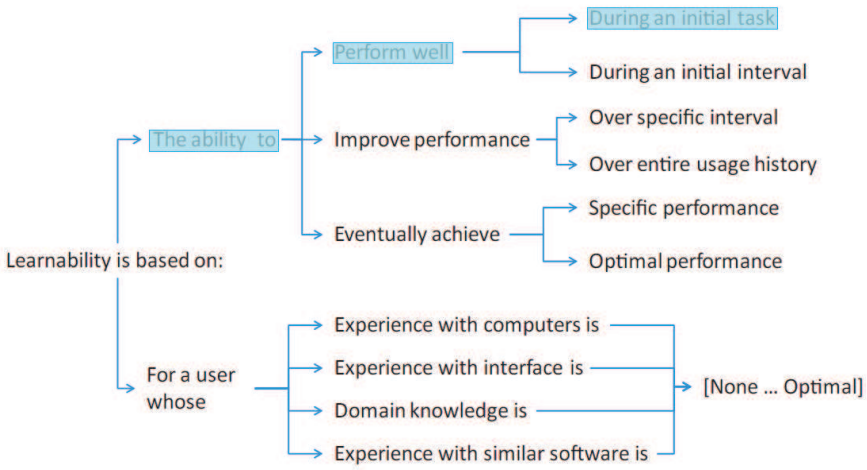

Figure 13 – Taxonomy of learnability definitions from Grossman et al. (2009)

We are not so much interested in the change of performance over time. Initial task performance on the other hand, is a good indication of the amount of complexity that is inherent to a certain degree of flexibility. The 'ability to' dimensions in the upper of the taxonomy can be complemented with dimensions to constitute a user definition. The attributes included are the user's experience with computers, similar interfaces, similar software, and the level of domain knowledge required. Note that both these dimensions and their purpose are similar to the ISO context of use attributes related to the user characteristics. Since the focus of this research is on initial task performance, it would be redundant to include the dimension about the users' UI experience. To summarize, the concept that we attempt to measure during the customer-evaluations can be described as:

> *"the ability to perform well during an initial task for a user unfamiliar with similar functionality and whose domain knowledge is optimal".*

3.5.1.5. Metrics

In order to obtain a relative usability score for each of the evaluated variability points, one or more metrics need to be set up for each of the defined sub variables. Note that one and the same metric can be a measure for multiple sub factors, while each sub factor can be measured by a combination of metrics. The main goal of the operationalization is that each sub factor is measured by at least one metric, and that all metrics (and thus all sub factors) together constitute a measure to gain insights in the usability concept defined for this research.

The efficiency can be measured by the first three metrics. The time participants require to complete a task is the most important measure for efficiency. The number of clicks and the time between inputs provide complimentary data which can be valuable for a better understanding of the time on task metric. For example, imagine that the average time on task for task 1 is much longer for product B than for product A. When the number of clicks required for fulfilling task 1 in product B is low in proportion to the time on task, this can indicate that participants spent relatively much time being unproductive – e.g.

participants were searching or thinking, instead of clicking. In both cases this would indicate a usability problem. When participants spent relatively much time on thinking, it can indicate that the task is experienced as being (too) complex (i.e. this would be an example of usability being negatively affected by the provided flexibility). However, when participants spent relatively much time on searching, this might be an indication of a GUI related usability problem (i.e. participants were not able to locate the required functionalities to fulfill the task). In this case one could conclude that the decrease in usability was caused by the GUI instead of the flexibility of product B. The 'time between inputs' metric can provide even more prove for the described situation.

The effectiveness is measured by the number of errors made during a task and the number of suggestions or tips the participant required to complete the task. As these two metrics can affect each other in many ways, they should be considered as a pair constituting one measure for effectiveness. For example, each suggestion provided by the coach can prevent the participant from making an error. Vice versa, participants may learn from making an error which makes help from the coach unnecessary.

Since our definition of learnability is based on initial task performance, we consider the combination of both the efficiency and effectiveness metrics suitable to measure learnability.

Table 5 – Usability metrics

#	Type	Description
1	time on task	Time until a user completes a certain task successfully (Nielsen & Hackos, 1993).
2	#clicks	The number of mouse clicks required by participants to perform a certain task.
3	Max time between inputs	The maximum time in seconds between keystrokes or mouse clicks during a specific task.
4	#help markers	The number of suggestions that a coach provided to a participant during a certain task.
5	#error markers	The number of errors made by users performing a certain task (Michelsen, Dominick, & Urban, 1980). An error is defined as any action that does not contribute to the desired result (Nielsen & Hackos, 1993).

3.6. Related Work

In order to position this research, an overview of the current state of the art within the research area of variability in multi-tenant environments is provided within this section.

3.6.1. Software Variability

Although software variability is one of the major architectural aspects – besides scalability and tenant isolation – critical to the success of multi-tenant software products, the concept of variability points was already introduced by Jacobson, Griss, and Jonsson, (1997) to increase software reusability. The relation between variability and reusability is confirmed by the fact that most research on software variability has been applied in the context of software product lines, which are families consisting of similar systems with many variations, created to increase software reusability (Bachmann & Bass, 2001; Bosch, 1999;

Coplien, Hoffman, & Weiss, 1998). Hence, our working definition of variability – *the ability of a software system or artifact to be changed, customized or configured for use in a particular context* – origins from the area of software product line engineering (van Gurp et al., 2001).
A taxonomy of variability realization techniques for software product lines is provided by Svahnberg et al. (2005). Though only a small proportion of these techniques are applicable for multi-tenant environments as only run-time bound solutions are applicable. Jansen et al. (2010) provide an overview of customization realization techniques (CRT) applicable within multi-tenant environments, based on the VRTs of Svahnberg et al. (2005). The CRTs are classifications of different variability types rather than patterns that can be used to actually solve a specific variability problem in multi-tenant context. However, the CRTs enrich the vocabulary on multi-tenant customization which facilitates the identification of variability patterns. Two case studies of multi-tenant applications are conducted in which some of the variability realization techniques are observed. Details about the CRTs are provided in the theoretical background section of this thesis.

3.6.2. Variability Modeling

Research on software variability is not only focused on how it can be realized, but also addresses the need to model the variable points within both software products and -architectures. The modeling of variability was – just like variability itself – first introduced within software product lines. To mention a few, Jaring and Bosch (2002) proposed a method to represent and normalize variability in industrial software systems whereas Keepence & Mannion (1999) use patterns, and Svahnberg et al. (2005) use feature diagrams to model variability patterns. A summary and classification on existing variability modeling techniques for software product lines is provided by Sinnema & Deelstra (2007). As is the case with variability realization techniques, only modeling techniques supporting run-time variability are useful within a multi-tenancy context.
Mietzner et al. (2009) describe how variability within multi-tenant SaaS applications can be managed and how knowledge on previous customizations made by tenants can be generated, by applying variability modeling techniques originating from software product line engineering. Knowledge on previous customizations can be used to create profiles or templates to increase the efficiency of the customization process. The authors use the Orthogonal Variability Modeling language described in (A. Metzger, Heymans, Pohl, Schobbens, & Saval, 2007; Pohl et al., 2005) to model variability while differentiating between external and internal variability (details are provided in the theoretical background). Montero, Peña, & Ruiz-Cortés (2008) present an approach for the modeling of run-time variability specialized to business-driven development (BDD) systems. The authors evaluate the modeling approaches of Svahnberg et al. (2005) and Gomaa & Hussein (2007) resulting in their own specialized approach, which should provide enough expressiveness for process-engineers – the key initiators of change, or evolution, within BDD systems. Besides a modeling approach, Gomaa & Hussein (2007) present run-time reconfiguration patterns enabling changing from one product line member to another on run-time. While this approach is an example of run-time variability, the approach is explicitly built upon the software product line philosophy and therefore not suited for multi-tenant SaaS applications.

3.6.3. Variability Maturity in SaaS

A competency model and a methodology framework designed to aid SaaS providers in the planning and evaluation of their service configuration and customization capabilities is presented by Sun, Zhang, Guo, Sun, and Su (2008). From 'completely standardized' to 'fully tenantized', five maturity levels based on the provided level of variance are defined. In addition, an overview on tenant perspectives on variability and four variability implementation strategies along with the realization costs in relation with the number of tenants are presented. Details on variability maturity levels are provided in the theoretical background section of this thesis.

3.6.4. Multi-tenancy Architectures

A large part of the sources related to both variability and multi-tenancy, present software architectures for the implementation of multi-tenant environments. While these architectures often describe how to address scalability and tenant-isolation, the implementation of tenant-based variability (most often in the form of configurability) is never omitted. This indicates that, within the current body of knowledge, variability is considered as one of the key aspects in the design and development of multi-tenant environments.

One such (high level) multi-tenant application architecture is presented by Chong and Carraro (2006). This architecture includes a meta-data service which is the primary means to facilitate tenant specific customizations in the four areas of user-interface and branding, workflow and business rules, data-model extensions, and access control. The approach is focused on the realization of a multi-tenant-efficient, scalable, and configurable SaaS application. Based on these three quality attributes, a SaaS maturity model consisting of four maturity levels is presented. SaaS applications at the second maturity level facilitate tenants to configure the application to make it fit in their business context. At the third and the fourth maturity level multi-tenant-efficiency and scalability are added respectively. As configurability is introduced at a lower level than the two other attributes, the importance of variability (in this case by implementing configurability) in multi-tenant environments is emphasized. Another contribution related to software variability are three methods (dedicated tenant database, shared database with a fixed extension set, and shared database with custom extensions) to implement an extendable data model which enables tenants to add tenant specific fields or entities.

Kwok et al. (2008) elaborate on their multi-tenant application for electronic contract management which facilitates several types of tenant-based customization. The authors extensively elaborate on what configurability options are provided to their tenants. However, only a high level architecture displaying the components involved in facilitating variability is provided. Except for the data model, which is customized by a shared database with fixed extension set, – as proposed by Chong and Carraro (2006) – any details on how these components implement variability is omitted.

The Force.com is a multi-tenant internet application development platform (i.e. Platform as a Service) hosted on the infrastructure of Salesforce.com (Weissman & Bobrowski, 2009). The platform enables external developers to build SaaS applications on top of the platform itself. However, most customers use the platform in conjunction with Salesfore.com, enabling extensive customization of the multi-tenant CRM product (Natis, Knipp, Valdes, Cearley, & Sholler, 2009). Weissman and Bobrowski (2009)

elaborate on the metadata-driven software architecture that is the foundation of the Force.com, and explain why this architecture is the premier choice for implementing multi-tenancy. Since the meta-data driven architecture of the Force.com enables the development of custom applications, it could be documented as one big variability pattern, just like any other architecture facilitating a PaaS (Platform as a Service). However, the meta-driven architecture described in this paper can be split up in a few variability patterns all implementing different kinds of variability. In addition, the architecture contains various techniques addressing performance, scalability or tenant-isolation issues while variability patterns focus merely on the implementation variability points. The architecture presented by Weissman and Bobrowski (2009) includes techniques facilitating custom extensions to standard data objects, the creation of custom data objects (including their fields and relationships with other custom or standard objects), the customization of the user interface (e.g. screen layouts, reports, data entry forms) and business logic (e.g. validation rules), custom security and sharing models (e.g. users, organization hierarchies, etc.), and the customization of application workflows. All these techniques can be classified as configurability as they implement variability points that enable tenants to customize various aspects of their application without the need for coding, but just by using the platforms point and click application development interface. In addition, the Force.com contains a metadata and web services API which can be used to access and modify the metadata and data model (business data) per tenant. APIs can be used to extend an application and therefore can be classified as customizability (contrasting configurability) providing much flexibility. The last variability point contained by the Force.com – also classified as customizability – is called APEX and can be considered as one of the most flexible run-time variability points in industry. APEX is an OO programming language similar to Java, executed on the Force.com platform enabling external developers to declare variables, constants, flow control statements (e.g. if, else, loop) transaction control operations, in order to construct routines that add custom business logic to various application events (button clicks, data updates, web service requests, et cetera). The main principle behind the metadata-driven architecture is that there is a compiled run-time engine (the PaaS core) that generates tenant-specific applications based on the metadata describing these applications. The compiled run-time engine is thus the static software developed by the PaaS provider whereas the actual applications that can be used by tenants are polymorphic as they are generated on run-time, based on the metadata describing the customizations made by the tenants themselves.

In (Bezemer et al., 2010; Bezemer & Zaidman, 2010), a multi-tenancy re-engineering pattern is presented which describes a lightweight approach for re-engineering existing single-tenant software into multi-tenant software. The goals of the re-engineering pattern are to migrate from a single- to a multi-tenant application while:

- only minor adjustments in the existing business logic are required;
- letting application developers unaware of the application being multi-tenant;
- clearly separating multi-tenant components to enable monitoring and load balancing.

Besides the possibility to share hardware resources for cost reduction and application and database instances for easier maintenance, the resulting application should possess a high degree of configurability concerning the workflow and look and feel of the application. To reach these goals, the

target (single-tenant) application is extended with an authentication, configuration and database component. Only the configuration component is of relevance for this research as it implements variability within a multi-tenant environment. The component facilitates four types of configuration including layout style, general configuration (e.g. personal profile details), file I/O (i.e. specification of file paths e.g. for report generation), and application workflow. Although the pattern is rather abstract and merely describes what components are added to the target application, the authors performed a case study at Exact which provides some implementation details. However, the configurability was not entirely implemented as the work flow configurability was marked as future work whereas the other configurability types are rather trivial and limited 'condition on a variable (Svahnberg et al., 2005)' implementations.

The challenges of implementing a native multi-tenancy architecture are addressed by the framework of (Chang Jie Guo et al., 2007). These challenges include security, performance, availability, manageability and configurability. In the concise subsection spent on the latter, the term customization point, which is another word for the concept of a variability point, is introduced. The authors present a lifecycle for run-time customization which is a high level process/architecture depicting the customization flow between actors (tenant-admin, tenant-user, developer and platform admin) and components (e.g. configuration wizards and repositories). Since run-time customization is often performed by a tenant-administrator, the use of configuration wizards with clear instructions and an intuitive UI are required to ensure the quality and consistency of self-service customizations.

3.6.5. Variability Frameworks and Architectures

In contrast with the multi-tenancy architectures described in the previous section, the related work elaborated on in this section has a pure focus on the realization of variability in multi-tenant environments.

Tsai et al. (2010) propose a multi-layered customization framework that describes an approach that SaaS providers can follow in order to create variable multi-tenant SaaS applications. The application has an architecture consisting of four logical layers (UI, process, service, and data) where domain ontologies are used to describe the objects residing in the layers and the relationships of objects both at layer and cross-layer level. The tenants can customize their application by selecting their required objects from the domain ontology on each level which enable tenants to build an application fitting in their business context. Tenants can select data-objects from the domain ontology to construct their customized data-model compose a workflow from the available services and compose their user-interface by selecting UI objects. To speed up the customization process, standard object templates are used. The authors also elaborate on the concept of variability granularity levels which is a fundamental concept in the multi-granularity framework proposed by Li et al. (2009) aimed to foster the customization process by clarifying the relations amongst objects at multiple granularity levels in the multi-tenant environment. The four identified variability granularity levels (parameter, object, cooperation, and coding) are compared on the amount of coupling, flexibility, scalability, and difficulty involved. For this research, the most important finding is the positive correlation between flexibility and difficulty. For example, variability implemented at object granularity is both less complex and less difficult to use by the tenant,

and to implement by the SaaS provider compared to variability implemented at cooperation granularity. This helps us to classify the variability patterns for the implementation of tenant-dependent reporting.

Nitu (2009) addresses the need for configurability in SaaS by providing an architecture supporting configurability within four different application aspects of an application. These aspects include user-interface, work flow, data, and access control, and are – excluding the access control aspect – quite similar to the logical layers defined by Tsai et al. (2010) and the granularity levels defined by Li et al. (2009). Besides the four application aspects, the distinction between configuration data and application data and between designers and users are the foundations of the presented architecture. Both designers and users are users working for a tenant. However, a designer is configures the application (edit mode: modifying configuration data) whereas the user performs tasks within the customized application (play mode: modifying application data). The differentiation between application data and configuration (meta) data is also applied in the architecture of the Force.com (Weissman & Bobrowski, 2009). The architecture is built up from an application module containing a designer and a user module. The designer module is used by the designer to configure the application in all four aspects mentioned whereas the user module is similar to the run time engine of the Force.com, which generates and 'plays' the applications on run-time based on the configuration data. While the fundamental thoughts behind the architecture are useful, the actual configurations are disappointedly limited as they are done based on pre-defined templates provided for each application aspect.

Mietzner (2008) proposed an XML format to define variability descriptors used to annotate the variability points in a multi-tenant application. The notion of the application template is introduced, referring to an application that is annotated with variability points. An application template is developed by the SaaS provider whereas customers are able to customize it by binding the variability points (described with the XML variability descriptors) resulting in a so called application solution hosted by the SaaS provider. Although the paper is focused on service oriented architecture based applications, the described approach should be generic to all multi-tenant applications. Based on the service component architecture (Beisiegel et al., 2007), a package format for SOA based application templates is proposed (Mietzner, Leymann, & Papazoglou, 2008). Variability is added to the SCA packages by means of variability descriptors (Mietzner, 2008) resulting in configurable multi-tenant SCA applications consisting of a fixed contribution – artifacts that are the same for all tenants – deployed once, and a tenant specific contribution which is separately deployed for each tenant.

The conceptual multi-tenant SaaS application platform presented by Kang, Kang, and Hur (2011) is based on the meta-data driven architecture presented in (Weissman & Bobrowski, 2009) resulting in the separation of business data and configuration metadata, and a run-time engine generating SaaS applications. The configurable aspects of the application are the same as those described by Nitu (2009) and in addition, an extra aspect called business logic configuration is introduced referring to all logic that handles the information exchange between the database and UI.

3.6.6. Variability Patterns

There is little work that explicitly combines the topics of multi-tenancy, variability and software patterns. The most relevant contribution is provided by Kabbedijk and Jansen (2012) which present a conceptual model describing how software patterns can play an important role in solving software variability

problems in multi-tenant environments. The concepts of tenant-based variability and variability patterns are elaborated and an approach for the documentation of patterns is provided. In an earlier work by Kabbedijk and Jansen (2011) provide a classification of different multi-tenancy and variability levels in order to create a common vocabulary amongst researchers and practitioners. In addition, the authors elaborate three variability patterns, namely Customizable Data Views, Module Dependent Menu, and Pre/Post Update Hooks. The Customizable Data Views variability pattern is the most relevant to this research as it describes a solution for tenant-dependent reporting. When this pattern is implemented tenants are enabled to save their preferences on how their data is presented (e.g. the sortation or filtering of data). The other patterns implement a dynamic menu based on a tenants modules and the possibility for tenants to alter their workflow by adding custom functionality just before or after an event. These variability patterns have been observed during two case studies.

A large majority of the work on variability in multi-tenant environments is focused on the realization of configurability. For this reason, the work of Müller, Krüger, Enderlein, Helmich, and Zeier (2009) is interesting as it addresses the customizability of multi-tenant SaaS applications. The authors propose three types of customizability within a SaaS ERP context based on the location at which the customizations are hosted and executed; desktop integration, user-interface customization, and back-end customization (more details on these classifications can be found in the section on tenant-based variability). In the context of back-end customization, the authors provide a technique referred to as dynamic instance composition (a non-explicit variability pattern) that can be applied to empower partners to inject custom business logic. A domain specific language similar to XML is used to define custom business objects whereas the data mapper pattern – similar to the description of Fowler (2003) – is used to hide the complexities related to data persistency. The generic part of the pattern is limited to a high level architecture whereas the provided details on the pattern are specific to the Ruby technology.

Walraven, Truyen, and Joosen (2011) present a generic architecture of a multi-tenancy support layer facilitating customizable component-based multi-tenant applications. The multi-tenancy support layer is built on top of a PaaS (Google App Engine is the PaaS used in their prototype) and consists of four major components that are interacting to enable feature based customization. The first component is a multi-tenancy enablement layer responsible for tenant isolation and the separation of meta- and application data. The other three components constitute the flexible middleware extension layer; the feature management facility provides an API that the SaaS provider can use to manage the variability of both the application and the available feature implementations whereas the configuration management facility provides an interface that tenants can use to manage their specific configuration. The last component is a feature injector – an extension of the dependency injection pattern (Fowler, 2004) – developed to facilitate run-time rebinding of variations based on the tenant-specific configurations.

4 Prototype

As part of the case study conducted at Exact, a variability point is implemented as prototype within Exact Online (the multi-tenant SaaS product of Exact). The leaders at Exact recognize the importance of variability within software products in general and multi-tenant applications in particular. One of their variability related interests is whether and how external SaaS products can be used to facilitate tenant-dependent reporting. The developed prototype is a variability point that demonstrates how tenants can use Google Drive Spreadsheets to build reports on top of the data service of Exact Online. In this section both technical and functional details about the developed variability point are provided. The parts of the variability point that can be abstracted into more widely applicable solutions are described as variability patterns in the next chapter.

4.1. The Case

The case study is conducted at Exact – a multinational software product company founded in 1984 and headquartered in Delft. In 2012, Exact had revenue of € 217 million and approximately 1700 employees. The unit of analysis is their multi-tenant SaaS application called Exact Online (EOL). At the moment of writing, more than 150 employees (still increasing) are dedicated to EOL and the product is available in the Netherlands, Belgium and the United Kingdom. In total, the product has more than 25k tenants of which around 90% are companies with less than 20 employees and around 40% are sole proprietorships.

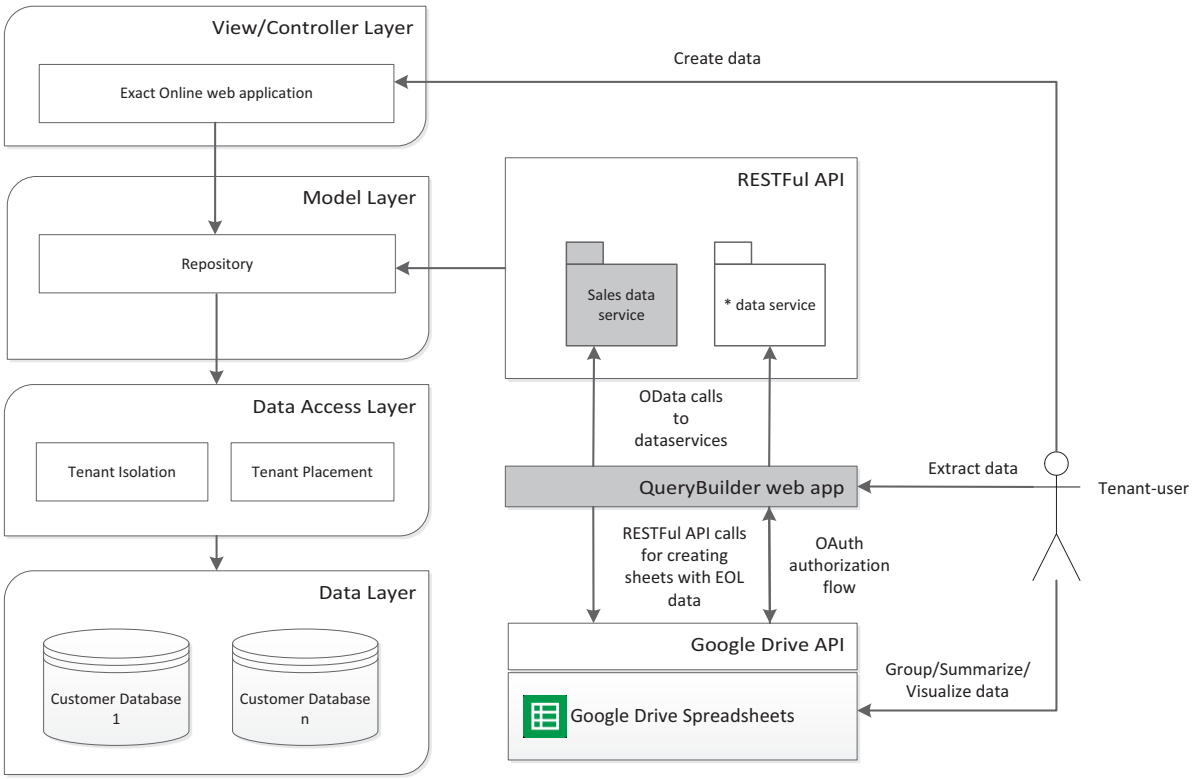

Figure 14 – Tenant-dependent reporting prototype implemented within the context of Exact Online

The high level architecture – i.e. the gross organization of a software system as a collection of interacting components (Garlan, 2000) – in figure 14 depicts the context of EOL after the implementation of the prototype. This architecture is by no means a complete representation of EOLs structure as it only includes components important for understanding the context of the case study.

As can be seen in figure 14, the EOL components are separated into multiple logical (i.e. not per definition physically separated) layers. The principle behind layers in software architecture is that a lower layer in the stack is not allowed to use the services – exposed via a public interface – of a higher layer (Bachmann et al., 2000). In addition to the geographic adjacency, the prescriptions of what layer may use or may be used by another layer are visualized by the arrows descending down the stack. These restrictions introduce advantages like portability, maintainability and reusability. For example, due to the separation of the business logic (model layer), from the application (controller) and presentation (view) logic – recall the MVC pattern (Krasner & Pope, 1988) – we were able to reuse the business logic of the repository for the construction of data services in the RESTFul API. Obviously, this restriction also increases the maintainability, as a change in the business logic only needs to be implemented once (in the repository).

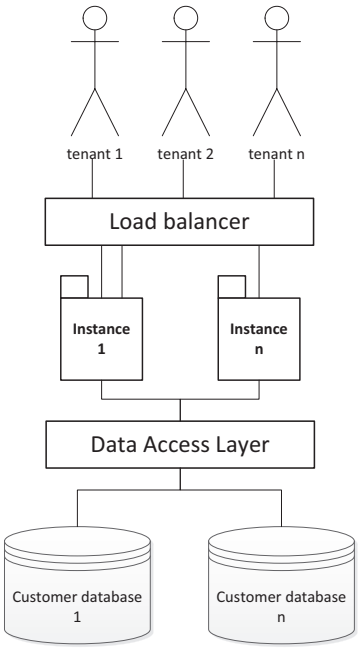

Figure 15 – The multi-tenancy architecture of EOL

EOL can be classified into the full multi-tenancy or level four maturity categories of Kabbedijk & Jansen (2011) and (Chong & Carraro, 2006) respectively. Besides a code instance, tenants also share a database, as depicted in figure 15. However, as the multi-tenancy maturity level four implies, EOL has a scalable multi-tenancy architecture. This means that although multiple tenants share a database and code instance, this does not mean that **all** tenants share the same database/code instance. The dividing of the load at code instance level (i.e. stateless sessions) is referred to as load balancing whereas the stateful distribution of tenants over databases in the database pool, is called tenant placement. In terms of the architecture in figure 14, the EOL databases reside in the data layer while the data access layer abstracts all issues regarding tenant-placement, tenant-isolation, and other tasks that should be transparent to the higher layers. To understand the tenant-isolation concerns that are abstracted to higher layers, imagine a scenario in which developers of EOL extend the repository with a new query. Due the data-access layer in between the data- and model layer, they do not have to think about which database to select and they do not have to include a tenant-id in their WHERE clause. The repository, a component containing the business logic of EOL belongs to the model layer, providing read/create/update/delete services to the EOL web application and the WCF data services of the RESTFul API. The EOL web application is implemented within the ASP.NET web framework. Since the aspx pages contain application and presentation logic, the V and C layers are merged within the architectural view of figure 14.

4.2. The TDR Prototype

For customer organizations (i.e. tenants) it is valuable to have the possibility to extract their data out of the multi-tenant SaaS products they use. These online business software products (like Exact Online) typically offer some variability for their reporting functionalities. However, these applications will never overmatch the reporting functionalities and possibilities of even pretty standard reporting tools like Microsoft Excel or Google Drive Spreadsheets – not to mention more sophisticated tools like Tableau Software (2013). When tenants have access to their own data, they are provided with maximum reporting flexibility. They can use external tools to conduct their reporting activities, and when tenants use more than one information system, it enables them to combine data from various information systems into one more valuable BI process.

As a first step in the prototype development, a new data service was added to the web API of Exact Online. However, as the typical tenants of Exact Online will not have the in-house knowledge required to utilize a web API, some sort of middleware needs to be developed that enables typical non-technical tenant-users to utilize the web API. For this reason, we have implemented the Query Builder web app, which demonstrates how such middleware can bridge the gap between a data API and successful tenant-dependent reporting.

4.2.1. The Data Service

In order to maximize the connectivity – i.e. to make it as easy as possible to connect with an application – a web API should apply standards wherever possible. As can be seen in figure 14, the web API built on top of Exact Online is RESTful, which implies that it follows the predominant REST (Representation State Transfer) web API design model. REST is an architectural style describing how distributed hypermedia systems should work (Fielding, 2000). While it is not a standard, it prescribes the use of standards such as HTTP and URI. A concise description on the constraints that a web API must adhere to in order to be RESTful, are provided in the Representational State Transfer pattern described in the next chapter.

When a data web API supports the Open Data Protocol ("OData," 2013) – describing a standardized way to consume and create data APIs – it automatically satisfies the REST constraints. OData builds on the architectural principles of REST, the Atom Publishing Protocol which is an HTTP based protocol for modifying web resources (hOra & Gregorio, 2007), and JSON (Crockford, 2006).

Since Exact Online relies on the Microsoft technology stack (ASP.NET, ISS, MSSQL) the Windows Communication Foundation (WCF) framework seems like the obvious choice for building a RESTful web API supporting the OData protocol.

During this case study, we have extended the RESTFul OData API of Exact Online with the Sales data service as part of the prototype development. Sales orders and accounts are the two topics that have been implemented within this data service. The sales orders topic is used within the tasks that are defined for the usability analysis. The second topic was implemented to be able to test whether the Query Builder Web App is able to deal with multiple topics in a data service.

4.2.2. The Query Builder Web Application

The second component constituting the tenant-dependent reporting prototype is the Query Builder (QB) web application. The goal of this application is to empower non-technical tenant-users to extract custom made selections of their data to Google Drive Spreadsheets utilizing the data web API of Exact Online. Figure 16 depicts the sequence of interactions between the components involved in a simplified use case scenario of the QB web application. The actor represents a tenant-user which is interacting with his web browser. Within the test environment used in the case study, the QB application is built as an extension of Exact Online. This means that an aspx page (containing the View and Controller logic in terms of the MVC pattern) was added to the source code of Exact Online containing all the JavaScript and HTML to run a pure client side web application. This means that in order to load this application into the web browser of the tenant-user, the client first needs to login at the Exact Online Web App. Forms authentication is used for this purpose. After a successful login, a response containing the QB app is sent to, and executed within, the web browser. Since the QB is a pure client-side application, the role of the Exact Online web application is finished once the QB code has reached the browser.

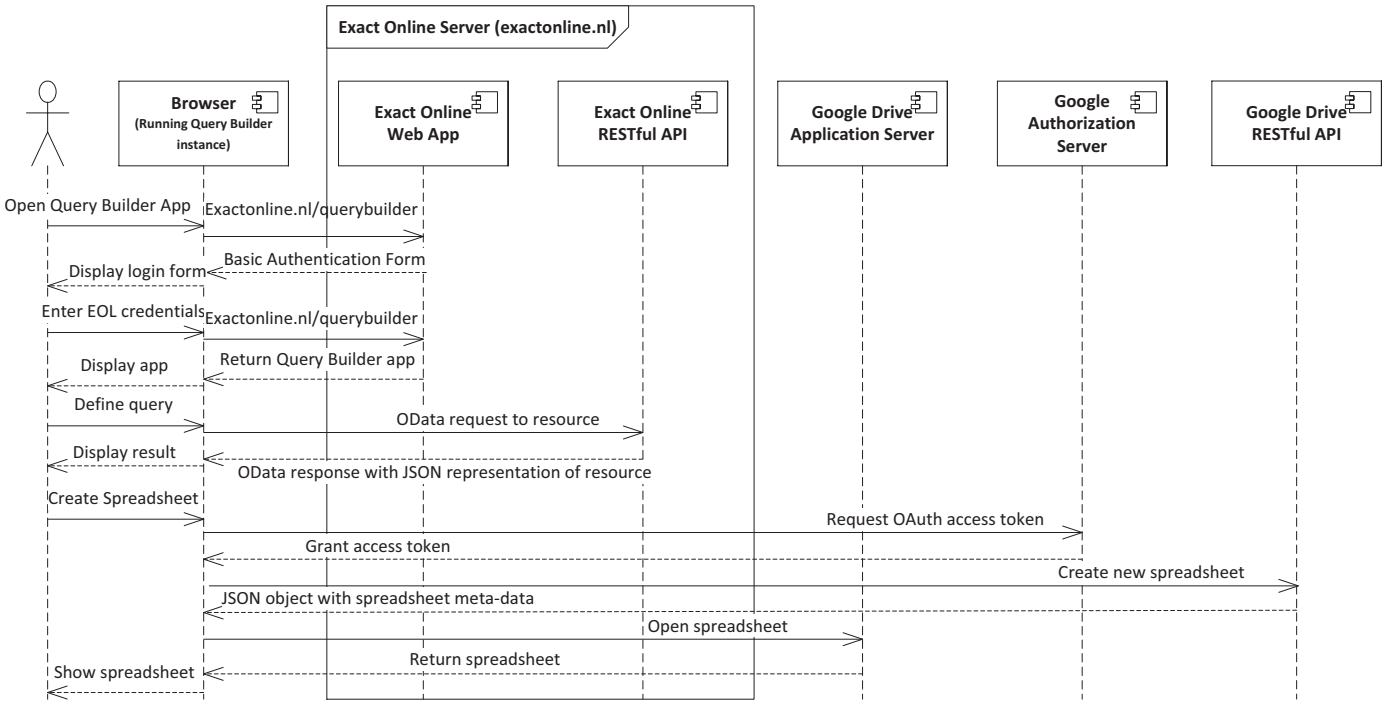

Figure 16 – Sequence of component interactions in a simplified Query Builder use case scenario

The tenant-user can now use the QB (of which the GUI is shown in figure 17) to select a topic, select columns available within this topic, apply sorting, and create filters, until the envisaged data set has been defined. When the topic is selected, the first OData request is sent to the EOL RESTful API. The response contains a JSON representation of the requested resource. In the example shown in figure 17, this resource is a set of sales orders. The first request is immediately followed by a second message, requesting the meta-data of the selected topic. Amongst others, this meta-data contains the data types

of the available columns. These are required to determine the applicable filter menu for each column in the data set (the QB includes a text, date and number filter) and the proper syntax in the query string (i.e. string values are quoted, number filters are not). Every time a sort or filter is applied or removed, or a column is (de)selected, the first 60 rows of the data set are fetched and shown in the data grid. Note that the number of 60 is also the maximum number of items that the EOL data API returns on one request. This server-side paging is a mechanism that should preserve the performance and availability of the data services. For the data-grid of the QB application, a preview of 60 rows suits well with the envisaged use case. However, when the tenant-user wants to create a Google Drive Spreadsheets file based on a selected data set bigger than 60 rows, let's say 6000, the QB needs to issue 100 requests to fetch the complete data set. Fortunately, as a true RESTful web service befits, the representations of the EOL API satisfies the *"Hypermedia as the engine of application state"* REST constraint by providing the URI of the next 60 rows in each representation. After all data is fetched, a POST HTTP request is sent to the Google Drive API in order to create the new Spreadsheets file. In the response of that request, the Drive API sends an object containing meta-data of the created file such as its URI, used to redirect the user to Google Drive Spreadsheets.

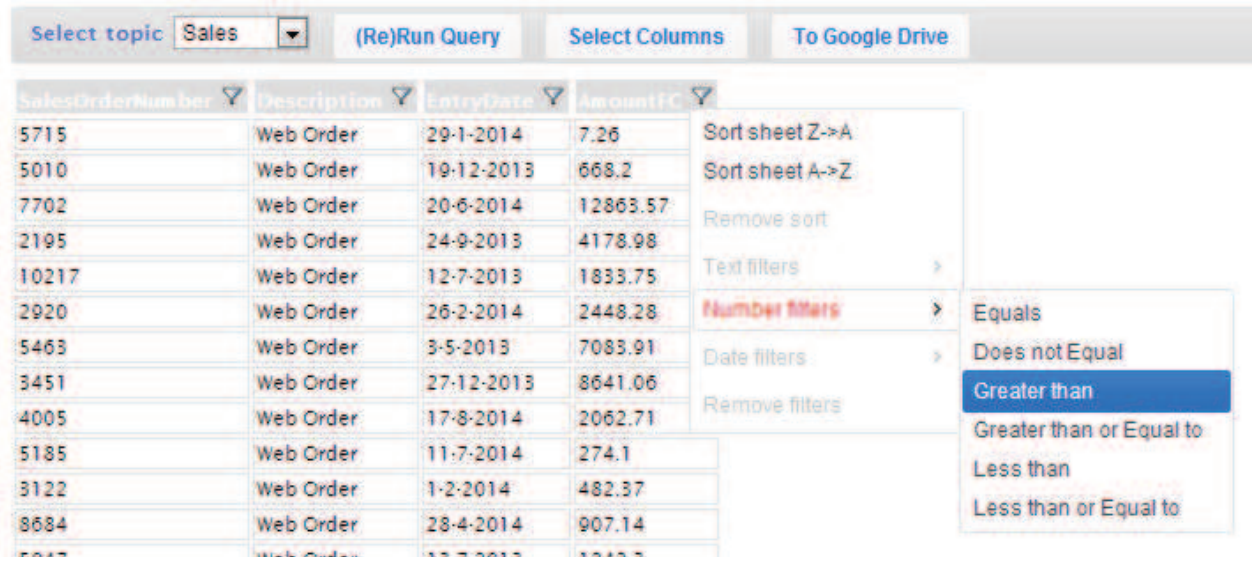

Figure 17 – A screenshot of the Query Builder Web App

All interactions shown in figure 16 (apart from the interaction between the tenant-user and the browser) are HTTP messages of which all except for the ones to and from the EOL Web App are sent asynchronously (i.e. once the QB is loaded, no page reloads are required). The jQuery (2013) JavaScript library is used as an AJAX (Garrett, 2005) framework and for DOM manipulation whereas the jQuery UI (2013) library is used to speed up the GUI development. Where jQuery is used to send requests to the EOL API, the JavaScript client library of the Drive API (provided by Google) is used to generate the requests sent to the Drive API.

4.2.3. The OAuth Flow

Only a small part of the communication in figure 16 represents the OAuth flow. Just before the Google Drive API (Resource Server) is used to create a Spreadsheets file in the Drive folder of the end-user (Resource Owner), an access token is requested at the Google Authorization Server. For the sake of readability, the scenario in figure 15 depicts the ideal scenario in which the end-user has already authorized the QB app – the Client – (i.e. during earlier usage of the app) to add Spreadsheet files to his Drive folder. Figure 18 depicts the abstract flow of the OAuth protocol as described in the standard proposal by Hardt (2012). When an end-user would start the QB for the first time, the QB redirects the user to a Google page (step A). When the user does not have an active session at Google, this page shows a login screen. After a successful login, the user is asked whether to authorize the QB to access his Drive folder and create files (this is regulated by Scopes defined by Google). Once the user grants authorization (step B), an access token can be requested at the Google Authorization Server (Step C and D). Once the access token has been received, the QB can use the Google Drive API (Resource Server) to create the Spreadsheets file, since this is within the scope of which authorization was granted earlier (step E and F).

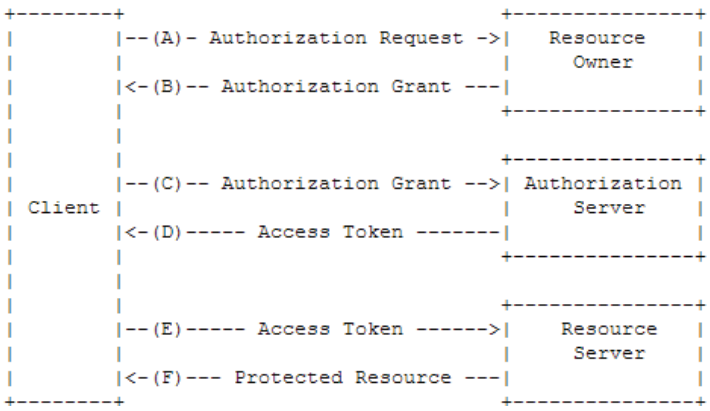

Figure 18 – The abstract OAoth protocol flow (Hardt, 2012)

4.2.4. The Spreadsheets API

As visualized in figure 16, the QB application communicates with the Google Drive API to create the spreadsheet file containing the data selected by the tenant-user. The Drive API can be used to add or remove both Google Drive files (e.g. the Spreadsheets MIME Type is application/vnd.google-apps.spreadsheet) and external file types (e.g. *.pdf or *.doc extensions) to the Drive account of a Google user. In addition to the Drive API, there is a specialized Spreadsheets API which can be used for Spreadsheet specific actions like adding rows or columns to a sheet in an existing file. Since, the QB application only uses the Drive API, it can only add data when a new file is created. The JSON is converted to CSV and then added to the body of the very same request that is used to create the file. By specifying the Spreadsheets MIME Type, the Drive API converts the CSV data to a Spreadsheets file. When the application used the Drive API for creating and the Spreadsheets API for adding the data, it would be possible to update the data in a sheet. Considering the scenario in which a user uses the QB to create a spreadsheets file with a given data set, where after he creates some pivot tables and graphs based on this data. The next month, this user wants to see the graphs and pivot tables with the data of the new month included. With the current version of the QB, the user has to create a new spreadsheets file with the QB app, and create new pivot tables and graphs. Even for a user with superb copy paste skills, this is a suboptimal solution.

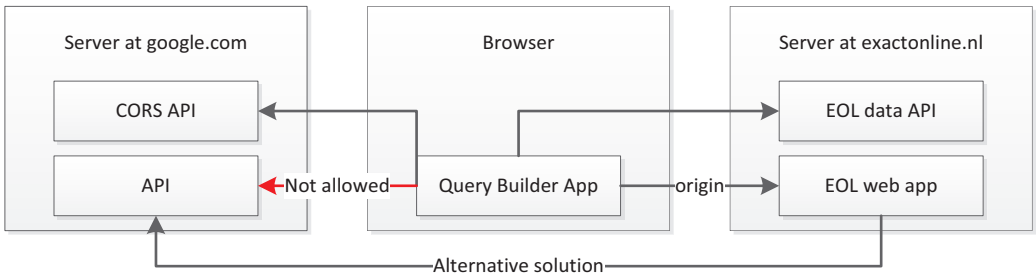

Figure 19 – The Same Origin Policy

As illustrated in figure 19, the same origin policy is a commonly applied restriction by user-agents (i.e. browsers) that prevents a client-side web application running from one origin, to obtain data retrieved from another origin (van Kesteren, 2010). An origin is constituted by a domain name (e.g. exactonline.nl), application layer protocol (e.g. HTTP), port number (80 is standard for HTTP).
The Spreadsheets API does not support CORS and no JavaScript client library is available.

4.2.5. Limited Capacity Google Drive Spreadsheets

For one of the tests conducted with the Query Builder, we have generated 8000 sales orders within a test account of Exact Online. The Query Builder app was used to select this data set of 8000 rows and 7 columns, and insert it into Google Drive as a new Spreadsheets file. Although this part of the test went quite perfect, creating a pivot table or graph based on this data set was impossible due to performance issues. This means that in the current practice, SaaS providers should only integrate with Google Drive Spreadsheets when a typical use case does not involve a lot more than 7000 rows. Fortunately, those tenants with relatively small data sets (recall that for Exact Online, 40% are sole proprietorships) are typically the ones that tend to use Google Drive Spreadsheets as an alternative to professional reporting tools. Larger customer organizations that generally have to deal with bigger data sets can directly use the data web API.
SaaS providers that want to integrate with Google Drive Spreadsheets, while their multi-tenant product typically involves large amounts of data, can decide to group data in the data services. Depending on the type of records and the grouping strategy, this can vastly decrease the amount of records that are inserted into drive. For example, imagine a company that has 100 sales orders a day. Grouping these sales orders per day would result in a hundredth of the original data set whereas a grouping per week or month will result in only 52 or 12 records on a yearly basis respectively. However, a liability of this solution is that this incurs extra work for the SaaS provider as there are many useful grouping strategies. Besides grouping based on periods, sales orders could be grouped per customer, business unit, product, product type, etc. Even when a SaaS provider implements much more than only the data services grouping per period, the tenants will always be limited in their reporting possibilities.
Summarizing we can conclude that SaaS providers should consider the use cases and in particular the expected amounts of data involved, before spending valuable development time on a Google Spreadsheets integration.

5 TDR Patterns

Besides the prototype discussed in the previous chapter, other variability points (i.e. functionalities or features that enable tenant-users to change the application so that it satisfies the needs specific to their organization) have been identified during a case study on EOL. To identify variability points, we have conducted interviews with product managers who are the functional experts of EOL. For additional details from the functional perspective, we had a test account and documentation of EOL at our disposal. For technical details we have interviewed software engineers that were responsible for the specific variability points. In addition, technical documentation and source code were available.

When the "pattern form" is used to describe a solution to a large or complex problem, decomposing the solution into sub solutions and writing them as part of a pattern language in which each pattern solves a specific problem within the shared context of the language, has advantages compared to stand-alone patterns. Breaking down a solution into smaller solution means that the solved problem is also split into smaller problems. Smaller solutions are easier to understand and smaller problems are easier to solve, which increases the simplicity. In addition, a single large solution – i.e. stand-alone pattern – tends to be specific to the circumstance which reduces its reusability whereas the parts of a decomposed solution may be reusable for various other circumstances as well. Finally, decomposing large solutions into smaller patterns constituting a pattern language makes it easier to reference to a specific part of the solution.

5.1. A Pattern Language for Tenant-dependent Reporting

5.1.1. Conventions

This pattern language adopts the style of Meszaros and Doble (1998) which means that pattern names are written in *italics*. Each pattern has a name by which it can be referenced and contains at least the context, problem, forces and solution elements. When useful, other elements such as the example are included as well.

5.1.2. Data Web API

Context

A software product company is offering a multi-tenant SaaS business application containing large amounts of tenant-specific data. This data would be of high value when included within the business intelligence activities of the application's tenants.

Problem

How can tenants be enabled to extract their data from a multi-tenant environment?

Forces

- The value of data increases when it can be shared, accessed by other systems or combined with other data.
- As the database – containing this valuable data – is shared by multiple tenants, providing tenants with access to this database is undesirable as security and privacy will be compromised.
- Even when multi-tenant applications have sophisticated reporting functionalities, data from one information system might be even more valuable when combined with other tenant-specific data (i.e. from other information systems used by the tenant) or external data. For example, a company might want to combine their data with numbers on the complete market to create benchmarks while another company which is offering products related to sun protection, wants to combine their internal data with data on the weather (both historical and forecasts).

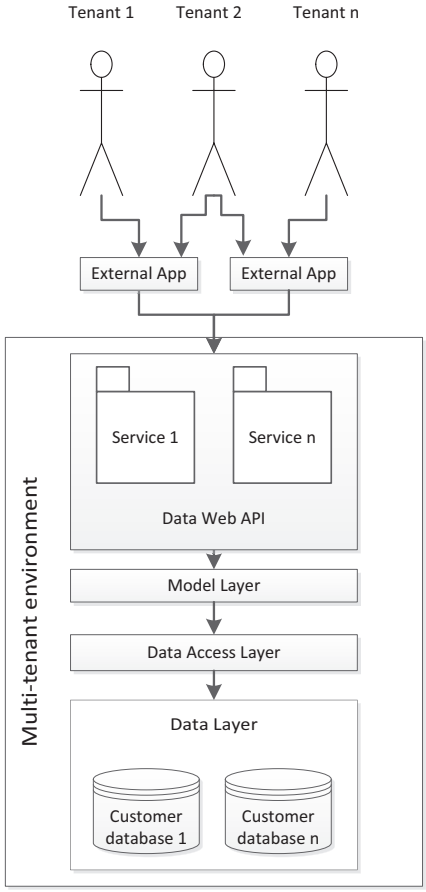

Figure 20 – Data Web API

Solution

Extend the multi-tenant SaaS environment with a web API (Application Programming Interface) providing data services which enable tenants to retrieve their operational data and use it within their business intelligence activities. Apply standards (e.g., OData, OAuth, XML, JSON) to increase the interoperability (i.e. enable third party applications to use the data services) and to limit the efforts required to use the data services to a minimum. Implement the API conform the architectural principles as described in the REST pattern to address its simplicity, visibility, scalability and performance.

Figure 20 depicts the Data Web API in the convenient context of a multi-tenant environment in which domain logic and data access logic are separated in layers. This is mandatory in order to preserve the maintainability and reusability.

The primary responsibility of an API is to map internal services to the outside world. Besides the actual data services, authentication and authorization are other indispensable examples of internal services that are exposed to external parties through the API.

Example

The *Data Web API* of Exact Online – described in chapter 4 – is an example of how this solution increases its value by implementing it following the *REST* principles and by combining it with *Data Middleware*. Reporting tools that can be used to connect with OData APIs – like the one of Exact Online – include Microsoft Excel 2010 (PowerPivot), LINQPad (2013), and Tableau Software (2013).

5.1.3. Data Middleware

Context

A software product vendor offers a multi-tenant SaaS business application mainly sold to small enterprises. The vendor already implemented a *Data web API* empowering their tenants to extract their operational data.

Problem

How can we enable non-technical tenant-users to extract data from a multi-tenant application?

Forces

- Small enterprises (often sole proprietorships) typically lack the technical knowhow (unless they have an IT related core business) required for utilizing a *Data web API*.
- Even when small enterprises do have the technical knowhow required for utilizing a *Data web API*, it still is quite labor intensive to connect a reporting tool to a data service.
- Not all reporting tools – and especially the ones that are free of charge (Google Drive Spreadsheets for example) – can be connected to a *Data web API.* Whether a tool can connect with a data API is also depending on the standards and protocols the API has implemented. There are reporting tools that can connect with OData APIs. However, the majority of the currently available data APIs do not use standardized protocols but instead provide developer documentation describing how to utilize their services. For this reason, the use of standard reporting tools to connect with *Data Web APIs* is limited to the ones that incorporated standards like OData (which is submitted to OASIS).

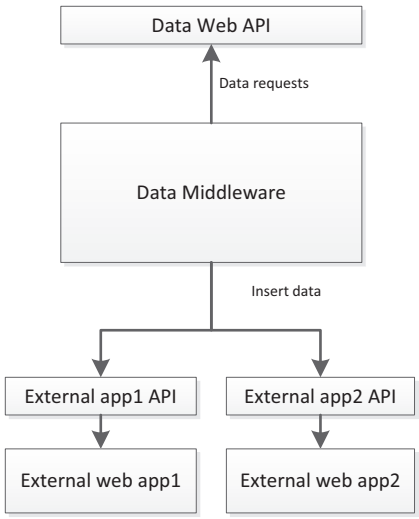

Figure 21 – Data Middleware

Solution

In order to enable non-technical tenant-users to extract a customizable subset of their data via the *Data web API* of the multi-tenant application of interest, a software product vendor can build a web application which functions as *Data Middleware* (figure 21) between a *Data web API* (the data source) and one or more third party reporting tools (the data destination). This middleware should provide a GUI which enables end-users to select the data they need, and export it into a third party tool in which they can perform their further reporting activities.

Example

The prototype variability point – i.e. the Query Builder web application – implemented during the case study on Exact Online is a good example of *Data Middleware* that enables non-technical end-users to utilize a *Data Web API.* End-users can select a topic (e.g., SalesOrders) and (de)select, filter, and sort columns, and subsequently export the defined data set into Google Drive Spreadsheets. A screenshot of

the – Excel like – GUI of the Query Builder is provided in figure 17, whereas the interactions with external components are depicted in figure 16.
As the goal of *Data Middleware* is to simplify the utilization of *Data Web APIs,* we performed a usability evaluation to test whether typical SaaS users are actually able to operate the Query Builder web application. From this evaluation we can conclude that users are indeed capable of effectively operating this example of *Data Middleware*. In chapter 6, details on the design and results of the usability evaluation are provided.

5.1.4. Abstract Query Builder

Context

You are implementing *Data Middleware* that provides a GUI through which users can construct a custom query. This application needs to support various data sources with their specific query languages.

Problem

How can *Data Middleware* generate queries for multiple types of data sources with corresponding query languages?

Forces

- In the process of constructing queries specific to one data source, a large part of the logic that needs to be implemented is required for constructing queries for other types of data sources as well. This generic or common part of the required logic needs to be implemented only once to ensure reusability. This also increases maintainability, as a change in the generic part of the logic needs to be changed in only one place in the code.
- To ensure extensibility, the code needs to be structured in such a way that newly supported query languages can be implemented in an easy way.
- A change that needs to be implemented for one query language (e.g. a syntactical change), should not affect other components in any way.

Solution

Enable the user to create a generic platform independent query using the GUI (like the one depicted in figure 17) of the *Data Middleware*. Depending on the selected data source (e.g., OData API, XML web service, SQL connection, etc.) the QueryBuilder decides what ConcreteQuery is returned to the GUI component.

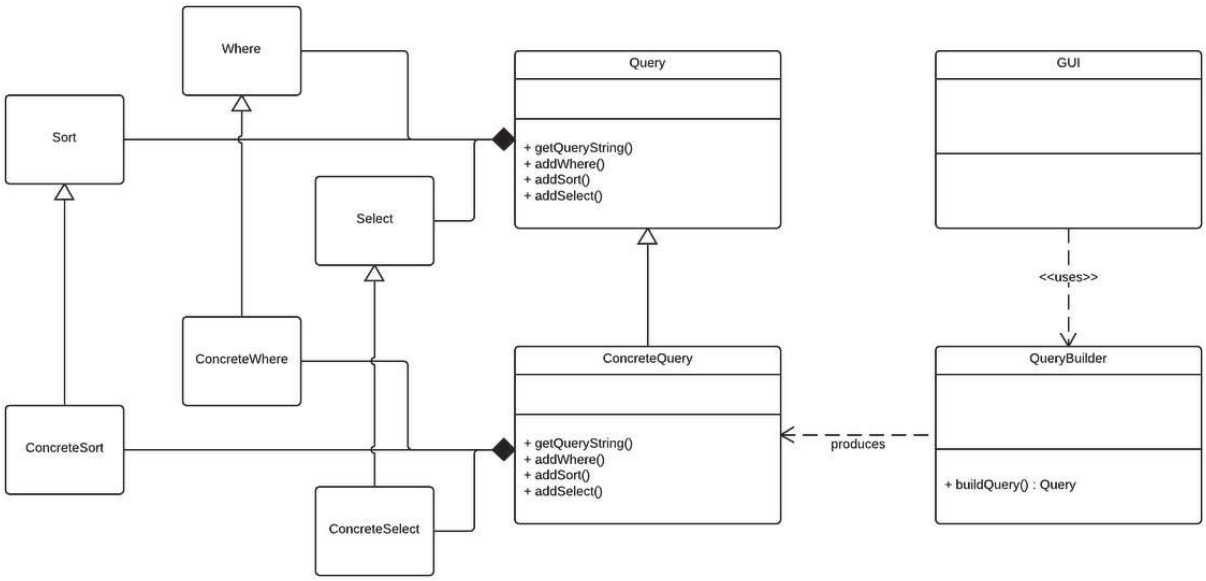

Figure 22 – Abstract Query Builder

Query is an (abstract) class and all its sub components – that is to say Where, Select, and Sort – are abstract classes as well. These classes need to be extended by concrete implementations which then inherit the common properties and functionalities amongst specific Queries and its sub components. Note that these can also be replaced by interfaces. However, as one of the key advantages of this solution is that common properties are centralized, abstract classes (or equivalents, depending on the programming language) are a better choice as they can contain these common functionalities.

5.1.5. Representational State Transfer

The REST architectural style – introduced and defined by Fielding (2000) – is currently the predominant way to structure web services. Besides the web API of Exact Online, some well-known examples of RESTful web APIs are those of Facebook, Twitter, Google Drive, Apache CouchDB (2013) and Dropbox (2013). Given the recurrence of this solution and relation with other patterns in this pattern language, the REST architectural style is suitable for documentation in pattern format.

Context

You are implementing a *Data Web API* on top of a multi-tenant environment to provide tenants with maximum freedom in how they use their data.

Problem

How can we implement a scalable and 'easy to use' *Data Web API?*

Forces

- As the goal of any SaaS provider is to sell its product to as many tenants as possible, the amount of calls to the *Data Web API* can grow significantly over time. To make sure that the web services will keep on performing even when the amount of tenants grows significantly, the *Data Web API* needs to be structured in a way that makes it scalable.
- One of the key benefits of 'opening' a multi-tenant environment by implementing a *Data Web API* is that it enables external companies to create applications on top of it. Extending its software ecosystem makes an application more interesting for tenants. However, to attract as many external developers as possible, using the API need to be as simple as possible.
- When a *Data Web API* incorporates widely used standards, it is far more likely that standard BI tools (i.e. not developed for your API specifically) will be able to connect to it.

Solution

When building a *Data Web API*, five architectural constraints need to be satisfied in order to enhance the quality attributes addressed by the forces of this pattern (i.e. simplicity, visibility, scalability, and performance).

Client-server

This constraint requires the existence of a client and a server. A **client** can send a **request** for a resource to the **server** which processes it after which a **response** containing a **representation** of the current **state** of the requested **resource** is **transferred** to the client.

Since the existence of both a client and a server is inherent to web services, this constraint is automatically satisfied when implementing a *Data Web API.* The client-server constraint is based on the separation of concerns principle, enhancing a systems portability and scalability (Fielding, 2000).

Uniform Interface

In order to increase simplicity and visibility, the interface between components should be as generic as possible (Fielding, 2000). For this purpose, four uniform interface constraints have been defined.

The first constraint is that every **resource** should have a unique **identifier** (in practice, the URI standard is applied for this purpose) that is managed by the authority that is also responsible for the meaning of the resource (e.g. the *Data Web API* instead of the clients). The identifier of a resource should not change, despite changes of the state or representations of that particular resource.

A second constraint is that resources can only be **manipulated** by clients **via their representations**, which should contain sufficient meta-data to instruct the client how this is realized.

The third constraint requires each message to be **self-descriptive**, i.e. a message must contain meta-data describing how to process it. For example, in the HTTP, Internet Mime Types (e.g. application/json) are used to describe how to parse a message.

As a fourth uniform interface constraint, it is required that clients can only issue state transitions (i.e. go from one resource representation to the next) by hypermedia. In other words, a client request must be based on one of the references contained in the current representation. A good example is the way one can browse through web pages (representations) by clicking on hyperlinks contained in each web page.

Layered-system

The layered system constraint requires the ignorance of the client whether it is connected directly to the server or to an intermediary component along the way. This means that intermediary components can be used for caching and load balancing purposes which increases scalability.

Stateless communication

When the information necessary to process a request is contained in the request itself, the stateless communication constraint is satisfied. By holding the session state at the client, multiple servers are able to process the request which enables flexible load balancing for increased scalability.

Caching

A systems' architecture must have a caching mechanism such that a response – containing a representation of a resource – can be labeled as (non-) cacheable. This enables clients or intermediary components to cache messages when appropriate, enhancing efficiency and performance.

In practice, RESTful web services use the same basic architectural mechanisms as the World Wide Web – worlds' largest RESTful system. The Hypertext Transfer Protocol (HTTP) is the application layer protocol used in both the World Wide Web and practically all RESTful web services, facilitating the required caching mechanism.

Example

Although REST does not prescribe the use of any technology, protocol, or standard, HTTP, URL, JSON, and XML are used in virtually all RESTful data APIs. The request methods of HTTP – i.e. POST, GET, PUT, and DELETE – match the with the CRUD data operations Create, Read, Update, and Delete, respectively. These operations can be executed on resources, which can be HTML pages, images, etc., but in the case of a data service, the resources are data objects or entities. A resource is identified by its resource identifier. In practice, the URL standard is used for this purpose. When a client requests a resource by sending a HTTP GET request with its unique URL, a representation of the resource is returned in the HTTP response. The Exact web API returns representations of entities in both the JSON and XML format, depending on the requested media type included in the HTTP request. An example meta-data attribute of a representation can be its media type, whereas an example meta-data attribute of a resource is its source link.

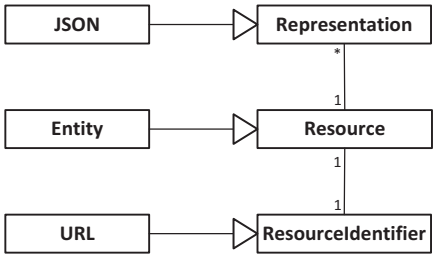

Figure 23 – Example REST implementation

5.1.6. Reusable View Components

Context

You are developing an information system in which GUI related patterns can be observed. In other words, certain view components – e.g., date pickers, item browsers, button bars, etc. – recur in multiple views. Considering this recurrence, you want to develop a framework that enables rapid development of views by reusing ready-made view components.

Problem

How can we facilitate the rapid development of views by using ready-made view components?

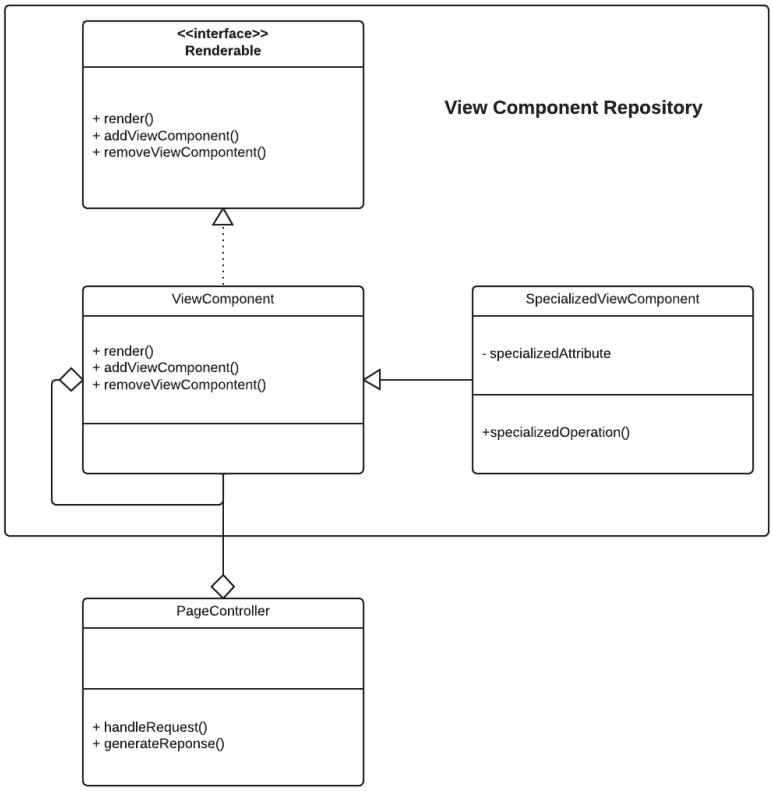

Figure 24 – Reusable view components

Forces

- By using ready-made view components, the time to market will decrease as development time decreases.
- The use of ready-made components takes away repeating and thus typically tedious from developers. This enables them to focus on more interesting and value-adding activities.
- Even when GUI patterns can be easily recognized, small page specific deviations can make it hard to capture them into a generic view component. For example, when it is observed that each page contains a button bar, a view component which can be used to generate a button bar should be developed. However, since the number of buttons, color, and triggered actions of the

buttons vary per occurrence, a certain degree of configurability needs to be added to the view component such that it supports all observed variants.

Solution

Identify all GUI patterns and determine for each whether it is worth the investment of developing a reusable view component for it. This should be determined based on the expected degree of future recurrence. As is depicted in figure 24, each ViewComponent implements the Renderable interface. Each ViewComponent can consist of other ViewComponents which all have the responsibility to render their own presentation (e.g. a piece of HTML and JavaScript code for web applications) and to handle the addition and removal of their sub components.

To address the force related to the complexity introduced by the degree of deviation amongst occurrences, variations on view components should be captured within specializations.

Example

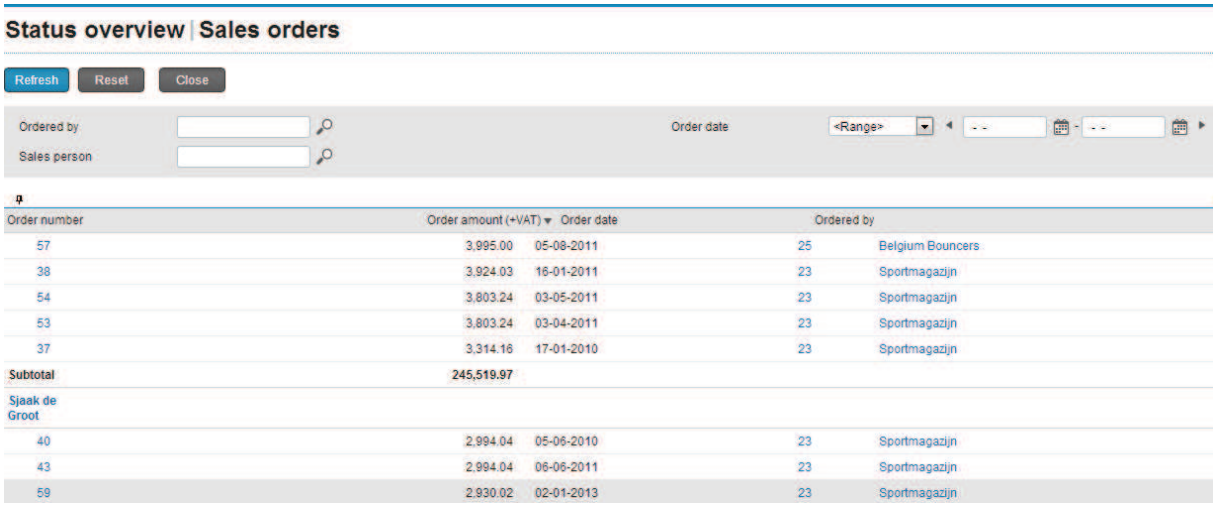

Figure 25 – A data overview developed using the reporting framework

Exact Online contains a reporting framework which implements the *Reusable View Components* pattern to enable developers to rapidly build any data overview in EOL, such as the one depicted in figure 25. The Buttons in the top left are contained within a ButtonBar whereas the Filters are part of FilterBlocks, which in their turn are part of a FilterSection. The FilterBlocks are used to group Filters that are related to each other, within the FilterSection. The table containing the data rows and columns is generated by the DataView class.

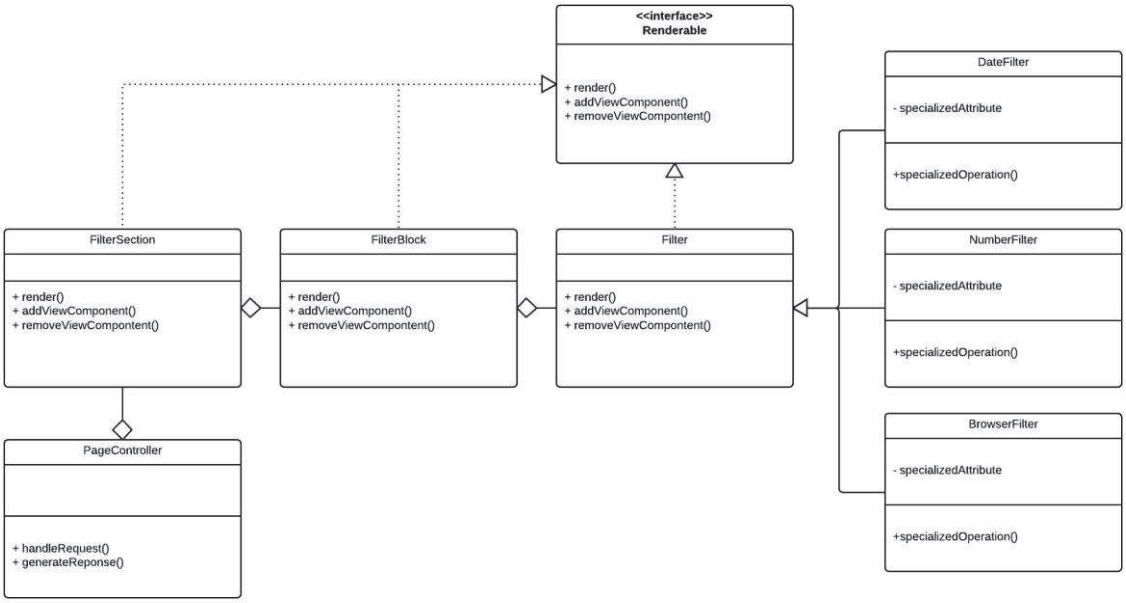

Figure 26 – Reusable view component implemented in the reporting framework

A small part of the implementation of the *Reusable View Components* pattern is shown in figure 26. When the PageController receives a page request, the render operation of the FilterSection is called which in turn calls the render operation of the FilterBlock and this recursive process continues until all sub components constituting the requested page have rendered their own view code. Note that various specializations of the Filter are created to deal with the deviations amongst filters while centralizing common functionalities.

In the reporting framework, the ViewCompontents are implemented by ASP.NET controls. Developers can use these controls in the aspx pages to rapidly constitute pages in the web application.

5.1.7. Tenant-dependent View

Context

You have implemented *Reusable View Components* resulting in a repository of ready-made components than can be used to construct views like the data view depicted in figure 25. However, now you want to use these ViewComponents to facilitate tenants with the functionality to create customized views.

Problem

How can SaaS vendors empower their tenants to create customized views?

Forces

- When SaaS vendors enable their tenants to customize views such that they only contain the ViewComponents that are required for that specific tenant, the usability of their SaaS product will increase.

- As tenants should not be able to create their customized pages by editing aspx files, a new technique is required to enable end-users to specify what ViewComponents should constitute their page.
- When end-users are provided with an increased degree of flexibility, the complexity of the customization process will increase as well.

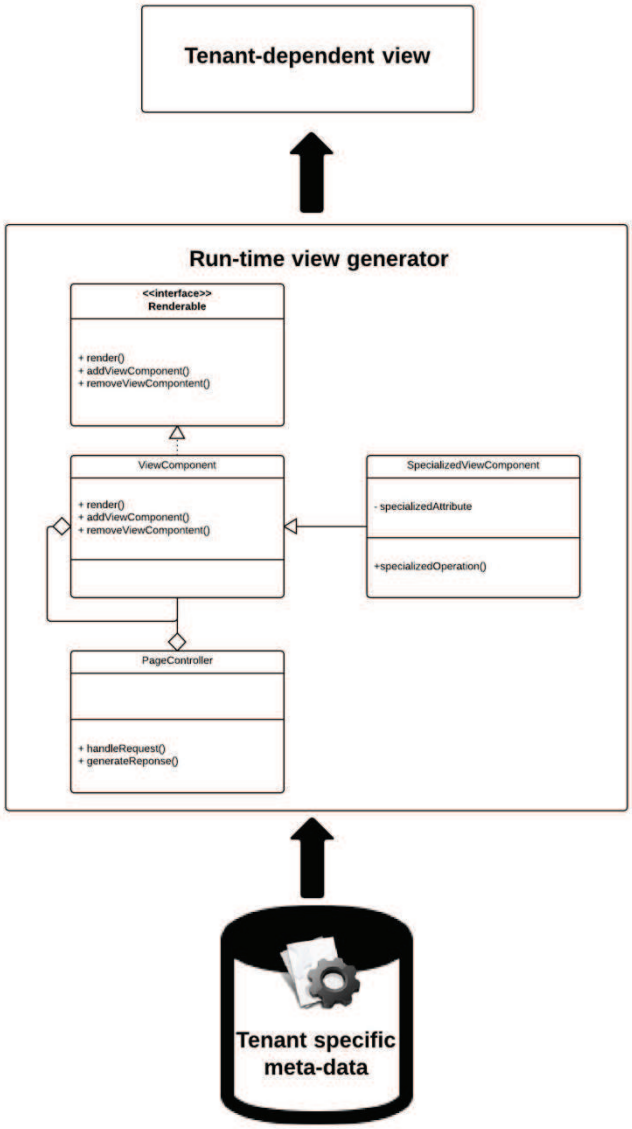

Figure 27 – Tenant-dependent view

Solution

Provide tenants with a GUI in which they can customize (certain) application views. This customization process results in tenant-specific meta-data describing how each view should be displayed for each tenant. The run-time view generator uses this meta-data to generate each tenant-specific view during run-time. In a high flexibility scenario, end-users can customize their pages by using the *Reusable View*

Components in a similar way as they are used by the developers – i.e. tenants are able to build pages from scratch. This type of variability is provided by the form builder of the Force.com (Weissman & Bobrowski, 2009). However, in most cases it is desirable to limit the amount of flexibility to configuring how ViewComponents are displayed, rather than what ViewComponents are displayed.

Example

The limited flexibility scenario as described in the solution part of this pattern is implemented within the reporting framework. Figure 28 depicts the customization GUI of the sales orders overview (figure 26) in which end-users can define what filters are shown in the filter section. The *Tenant-dependent View* pattern is quite abstract as it does not provide details on the organization of the tenant-specific meta-data. In Exact Online, this meta-data concerning the view configurations is stored in XML format in the user-settings tables contained in the customers databases.

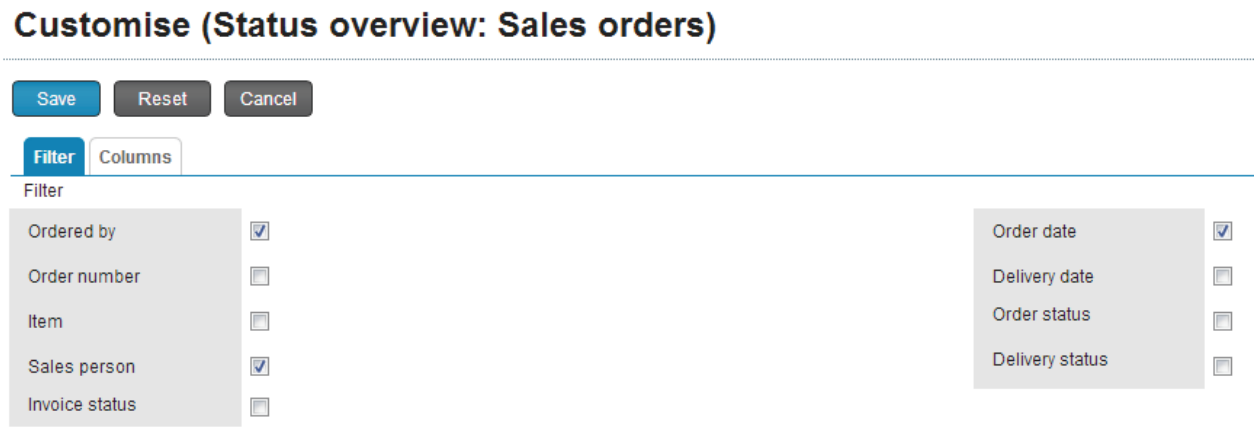

Figure 28 – Customizing the reporting framework

5.1.8. Customizable Authorization Model

Context

You are offering a multi-tenant information system which contains various kinds of resources and implements *Reusable View Components*. Examples of resources include modules, pages, view components, and data sources. A tenant typically groups a number of users that have various responsibilities and tasks in their employers' business processes. This is reflected in the resources that they require for fulfilling their tasks.

Problem

How can you enable tenants to customize the rights users have on specific application resources?

Forces

- The ability to provide users merely with the resources they require to fulfill their tasks is an ability that provides the potential to increase the usability. When a view contains redundant information and functionalities, user's task performance will typically decrease as they need to spend time searching for the resources they actually require.
- Information systems can contain information that tenants want to hide for parts of their employees.
- When rights are granted on a varying level of granularity resources, it becomes more complex to determine how a page should be shown to a specific end-user – especially due to low granularity resource authorization.

Solution

Provide tenants with a GUI in which they can define roles to assign rights on resources. As shown in figure 29, a Resource is an abstract concept referring to concrete types of resources like – amongst many others – a DataSource or ViewComponent. In Exact Online, four types of Rights exist, that is to say View, Create, Update, and Delete.

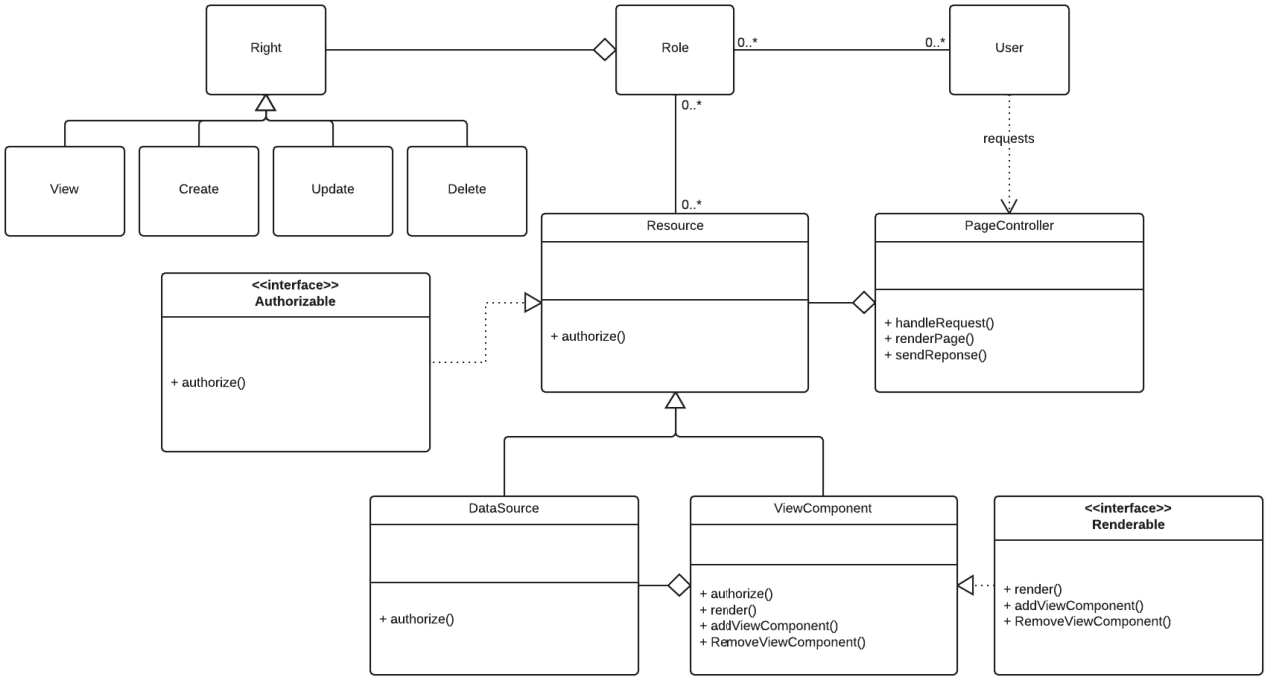

Figure 29 – Customizable authorization model

This solution addresses the force related to the complexity inherent to low granularity resource authorization by making the Resources responsible for their own authorization. When a User requests a page, the PageController calls the render operations of the ViewComponents that constitute the requested page. Each ViewComponent determines for itself and its sub components – as explained in the *Reusable View Components* pattern – whether and how they are represented on the page, based on the User authorization. Resources that are not ViewComponents are still contained within a

ViewComponent. For example, when a page containing a data overview of accounts is requested, this page contains a DataView – a specialization of ViewComponent – which is responsible for representing a DataSource containing the accounts data set. Subsequently, the DataSource determines what columns should be shown based on the Roles of the User requesting the page. In figure 30, an EOL GUI is shown in which tenants can assign roles on the address data of accounts. In this case, six different Roles have either View or Update and View rights on this Resource.

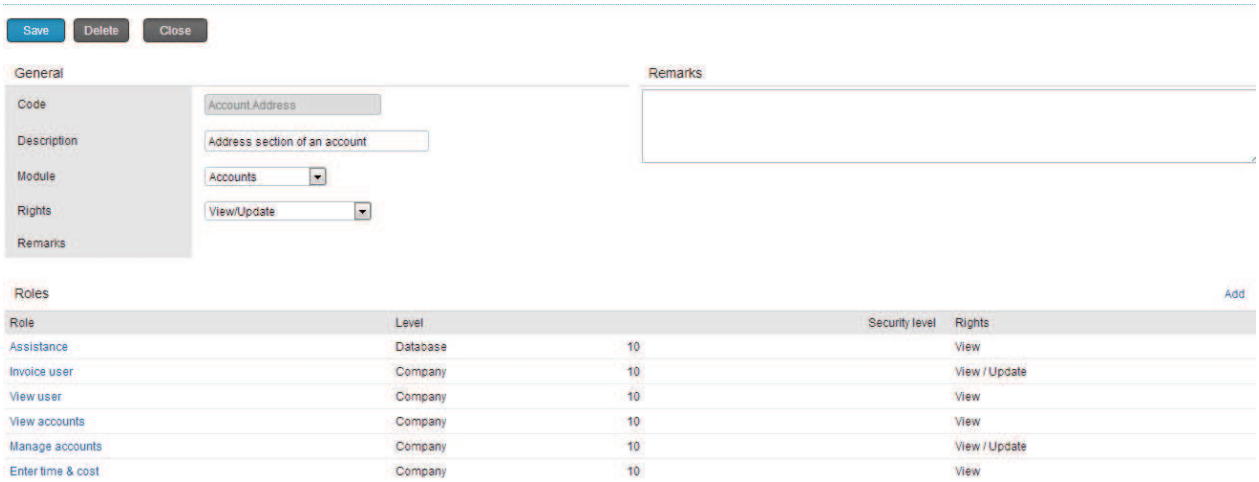

Figure 30 – Example of customizable authorization model in EOL

As can be seen from the example in figure 30, the described solution for a customizable authorization model provides much reporting flexibility as authorization is managed on a very low granularity (it is possible to grant rights on field level – e.g. Accounts.Address). The authorization model can be customized such that two customers that have rights to see the accounts data overview can still have a different representation, when one user has view rights on the address information while the other has not. This variability is not only advantageous from a security perspective, but also for the usability provided to users as all redundant resources can be hidden which reduces complexity.

6 Usability versus Flexibility

The implementation of variability within multi-tenant environments results in a software product that – despite the fact that one code instance is shared by multiple tenants – can be adapted to fit within tenant specific contexts. Besides the economic advantages for the SaaS provider (a larger market can be addressed as a wider range of requirements are satisfied), variability is advantageous for tenants as well, since more of their requirements are supported. However, one of the pitfalls of variability is that customization or configuration processes become too complex for the typical end-users. In order to address the hypothesized tradeoff between the variability (i.e. flexibility) and usability (i.e. simplicity), a usability evaluation on reporting variability points providing a varying degree of flexibility is performed. First we discuss the reporting variability points (VPs) which are evaluated on their usability and why and how these VPs differ in the amount of flexibility they provide. Then we describe how the variability points are evaluated whereupon we elaborate on the results of this usability study.

6.1. Reporting Variability Points

Exact Online (EOL) contains various kinds of VPs which enable end-users to customize the way data is represented, what data is represented; how and what is exported; et cetera. This data varies from a simple list of sales orders to the financial statements such as the balance or the income statement – which also can be customized to some extent. However, in most cases the VPs within EOL are rather limited in comparison with the flexibility provided by the Query Builder web application (QB).

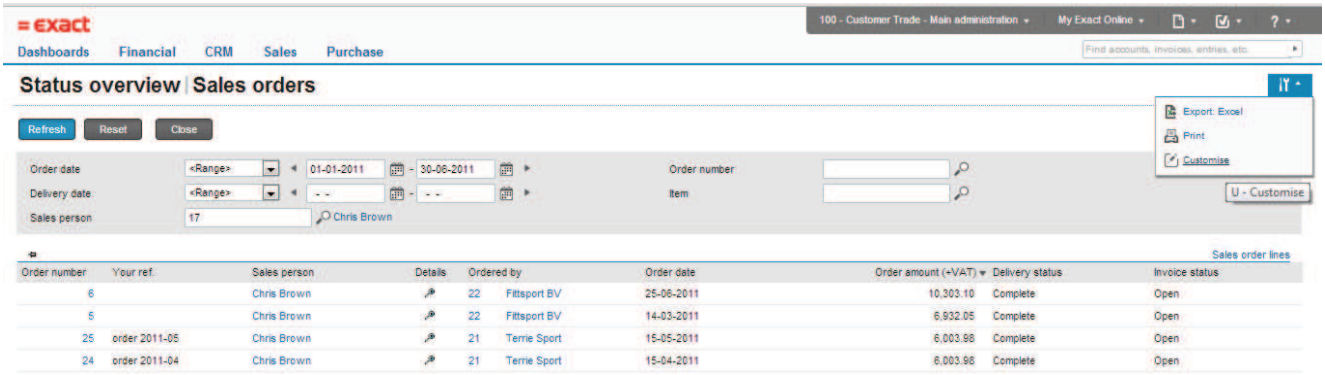

Figure 31 – The reporting framework in EOL

A vivid example of such limitation is provided by the reporting framework, which is used to display almost every data overview in EOL. In figure 31, the tenant's sales orders are listed by the reporting framework. Above the data grid, a few filter options are displayed. In this screen, users are limited to filtering and sorting the data. However, at the right top of the screen a menu containing the customization button which brings the user to the customization menu shown in figure 32. This menu can be used to add and remove available columns (i.e. attributes, fields, etc.) to and from the data grid. In addition, the column style and order can be changed and columns can be dragged into the group by section.

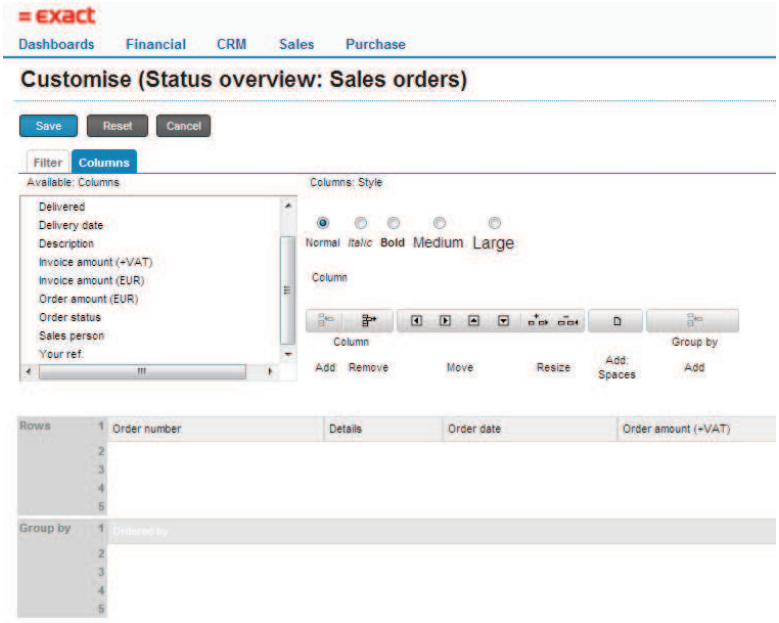

Figure 32 – The customization menu of the reporting framework

The reason that the reporting framework is considered to be less flexible than for example the Query Builder web application, can be illustrated by the filter functionality which is also shown in figure 31. In the reporting framework, only one filter per column can be specified and there are also columns that cannot be filtered at all (e.g. Order amount). For example, it is not possible to select the sales orders of both customer A and customer B. It is only possible to show either one of them or all of them in one overview. As shown in figure, the QB allows the user to add multiple filters on one column, and in addition, more types of filters can be added. The reporting framework is limited to the 'equals' filter whereas the QB (figure 33) allows, depending on the column type, various other types of filters such as 'greater than', 'less than or equal to' (for numeric fields) 'contains', 'ends with' (for text fields), 'Before date' and 'In month' (for date fields).

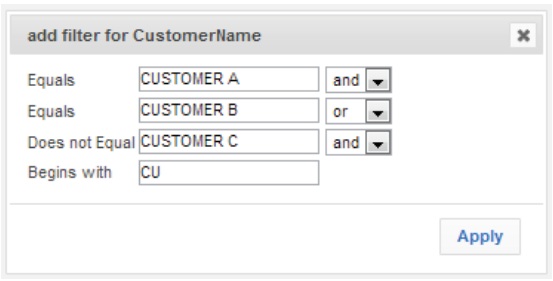

Figure 33 – Multiple filters on one column in the QB

Another example of difference in the flexibility provided by VPs is illustrated by the pivot analysis VP in EOL (figure 34) and the pivot analysis of Google Drive Spreadsheets. In EOL, graphs and pivot tables can be created, and consequently be added as a widget to a customized dashboard. The fields that are used in the rows, columns (both have a maximum of three fields), and values (maximum of four fields per cell)

can be specified and various filters can be applied. Despite the fact that the customizable pivot widgets are a nice example of reporting variability, this VP is limited in the aggregation operators that can be applied, since all fields that are added to the value cells are summed. Aggregation functions like average, min, max, or count cannot be used. The same goes for the 'group by' functionality provided by the reporting framework. When sales orders are shown grouped per sales person, the total order amount (SUM) of that sales person is shown, though it is not possible to add other aggregation functions to the report.

Report | Filter | Layout | Data

Financial year	2010	2012	2013	Total
Account	Amount	Amount	Amount	Amount
-	102,452.92			102,452.92
ABC Producten & Diensten B.V.		4,117.16	2,228.93	6,346.09
Fam. de Vos	3,020.17	5,993.40	6,693.39	15,706.96
Fam. Groot		2,090.84	7,494.21	9,585.05
Fam. K. de Vries		3,357.10	2,890.91	6,248.01
Fam. Schutze	1,510.92	4,325.21	9,336.36	15,172.49
Total	106,984.01	19,883.71	28,643.80	155,511.52

Figure 34 – Customized pivot analysis widget in EOL

In the graph builder present within EOL (figure 35), various kinds of charts can be created based on a set of predefined topics (e.g. revenue, costs, accounts payable, etc.). Within these topics, graphs can be customized – amongst others – by selecting a specific time span, selecting the values shown in the graph (e.g., for the revenue topic values include gross margin, revenue, cost of goods, etc.), and by configuring how to group these values. For example, the chart in figure 35 groups the values revenue and gross margin per financial year and period while these values could also have been grouped per general ledger (GL) account, cost unit, or country.

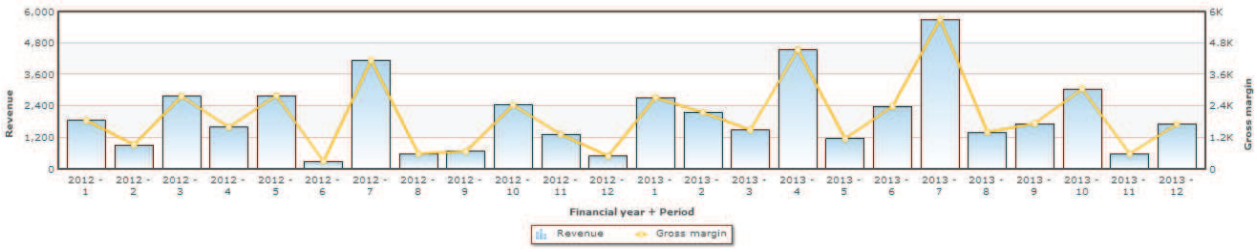

Figure 35 – Customized chart widget in EOL

Despite the fact that EOL enables users to customize graphs on the various mentioned aspects, again the possibilities are far more limited than it would be when a graph was created using QB combined with Google Drive Spreadsheets. For example, with the functionality depicted in figure 35, it is not possible to create a graph showing the revenue per sales person. And as another example, when grouped per country, it is possible to show the total sales order amount (revenue) of that country, but when we want

to know the average sales order amount per country, we have to switch to the more flexible solution such as the QB usage scenario.

6.2. Usability Evaluation Protocol

The *concurrent thinking-aloud* protocol is the predominant data collection method for usability evaluation (Van den Haak et al., 2004). With this approach, participants are verbalizing their thoughts while performing predefined tasks using the evaluated artifact. In this research, we evaluate multiple artifacts. The usability of three reporting variability points within EOL is compared with that of the QB in combination with Google Drive Spreadsheets.

Other data collection methods include the *retrospective thinking-aloud*, which differs from the concurrent thinking-aloud in that the thoughts are verbalized after the tasks have been completed (Van den Haak et al., 2004). With the *constructive interaction* method (i.e. co-discovery), two participants work together on a task while verbalizing their thoughts. In the *question-suggestion protocol* – which is based on the *question-asking protocol* – participants can not only ask questions to the coach, but in addition, the coach is free to give advice (Grossman et al., 2009).

The study of Van den Haak et al. (2004) compared the concurrent- and retrospective thinking-aloud and the constructive interaction protocols on their effectiveness for identifying usability problems. Although no significant differences in their effectiveness have been found, the concurrent thinking-aloud protocol could be preferred considering some practical advantages. First of all, the constructive interaction protocol requires twice the amount of participants to gather the same amount of data. In addition, the retrospective thinking-aloud session costs twice the time of a concurrent thinking-aloud session, as the participants have to comment (i.e. think aloud) based on a recording of their earlier performance.

However, using the question-suggestion protocol proved to be more effective in identifying usability issues compared to the thinking-aloud protocol (Grossman et al., 2009). In addition, more data on the usability measures can be collected in a shorter amount of time. For example, consider a task consisting of three consecutive activities constituting a process. When the participant has problems with completing activity one, we would not be able to collect data about activity two and three. However, when the coach would have been allowed to provide help or suggestions, the participant would have been enabled to proceed with the second and third activity as well, enabling us to identify more problems. To summarize, the concurrent thinking-aloud protocol proved to be more efficient than the retrospective thinking-aloud and the constructive interaction protocol. However, the question-suggestion protocol has proven to be more effective and efficient in the identification of usability issues in comparison with the concurrent thinking-aloud protocol. For this reason, the question-suggestion protocol is applied within this study.

6.2.1. Selection Criteria

In the usability section of the theoretical background, the concept of usability is defined and operationalized. We have defined three usability sub factors (learnability, effectiveness, and efficiency) which are measured by a total of five metrics. However, an important part of the definition of usability is the context of use, with the user characteristics as most relevant factor. The selection of participants for

the usability evaluation is based on this context of use description, resulting in the following selection criteria:

- Participants should have sufficient financial/bookkeeping knowledge to complete the tasks;
- Participants should have sufficient system experience (e.g. familiar with browsers and spreadsheet software like Excel or Google Drive Spreadsheets) to complete the tasks;
- Participants should be unfamiliar with the evaluated variability points (i.e. no existing users of Exact Online);
- Participants should not be technical experts (e.g. no BI specialists etc.);
- Participants should have sufficient knowledge of the English language, as this is used in the GUIs of the VPs.

6.2.2. Survey

After they have completed the usability evaluation session, each participant is required to fill in a small questionnaire containing questions to obtain subjective measures of the quality of the GUIs of the variability points. This is done to find out whether the GUIs affected the usability. Note however that the GUIs of all variability points are quite similar as they are all part of the same system. An exception to this is the Query Builder prototype, though its GUI is quite similar to that of the VPs in EOL. As another validity measure, the questionnaire contains some questions related to the participant selection criteria listed above.

6.2.3. Data Collection

In order to extract and record the required data from the usability evaluation sessions, the Morae (2013) usability testing tool is used. It enables us to capture the screen, the verbalized participant thoughts (and the conversations between coach and participant), and the facial expressions of participants, in one data stream. This has the practical advantage that the observer does not need to watch the usability evaluation session in real time, which means that only one participant and one coach are required to perform one usability evaluation session. In addition, as the coach does not have to make notes, he is able to fully concentrate on coaching and eliciting verbalized thoughts from the participant which enhances the quality and amount of data captured.
The time participants required to complete a certain task, the number of mouse clicks per task, and maximum time between inputs per task are quantitative metrics that are automatically calculated by Morae after the observer marked the start and end of each (predefined) task in the timeline of the recording. During the analysis of each usability evaluation session, error and help markers are added to the recording. Those markers are the input for the help and error metrics. For later analysis, the marker is labeled and a short description is provided.
To summarize, we use the Morae usability analysis tool to support data collection during the evaluation sessions as it enables us to collect data in a centralized and aggregated fashion, facilitating more efficient and effective usability analysis.

6.2.4. User Tasks

As described in section 6.1, we have identified variability points within EOL that facilitate usage scenarios that are also facilitated by the Query Builder web application in combination with Google Drive Spreadsheets (hereinafter referred to as QB). In addition, we have discussed why the VPs of EOL provide less flexibility than the QB does. The overlap in the possible usage scenarios and the demonstrably difference in flexibility enable us to identify the possible relationship between flexibility and usability. Based on the VPs of EOL, three tasks (each with a business case as context) have been defined. Participants performed each task twice, once with a less flexible VP of EOL and once in the more flexible QB. By measuring the task performance of the participants, there are three opportunities in which the impact of flexibility on usability can be observed.

The objective during the first task was to create an overview of the sales orders of one specific sales person within a certain time span. During the second task, participants had to create an overview of sales orders issued by one specific customer within a certain time span grouped by sales person. The objective of the last task was to create a chart showing the total revenue per period (year + month) within a specific time span.

Note that the complexity is increased with each task. The first task only involves filtering and sorting a data set, task two introduces the grouping of data and the aggregation of values (e.g. total order amount per customer) while the last task involves all the previous but now the grouped and summarized data needs to be graphically represented in a chart. In EOL, both the first and second task is conducted using the reporting framework (shown in figure 31). However, in order to (amongst others) group data, the second task involves the customize menu as well (shown in figure 32). The third task is performed using the graph builder (shown in figure 35). In the QB scenario, the first task is performed only in the Query Builder web application. In order to complete the second and third task, the participants also require the functionalities of Google Drive Spreadsheets.

6.3. Results

Eight participants performed three tasks in both the relatively limited EOL and the relatively flexible QB, in order to observe whether a relationship exists between flexibility and usability. Figure 36 depicts the average number of minutes participants required to perform a certain task in a certain product. Although we can observe that each of the three tasks took more time to complete using the most flexible VP of the two (QB), only the third task shows a truly large difference between the two VPs.

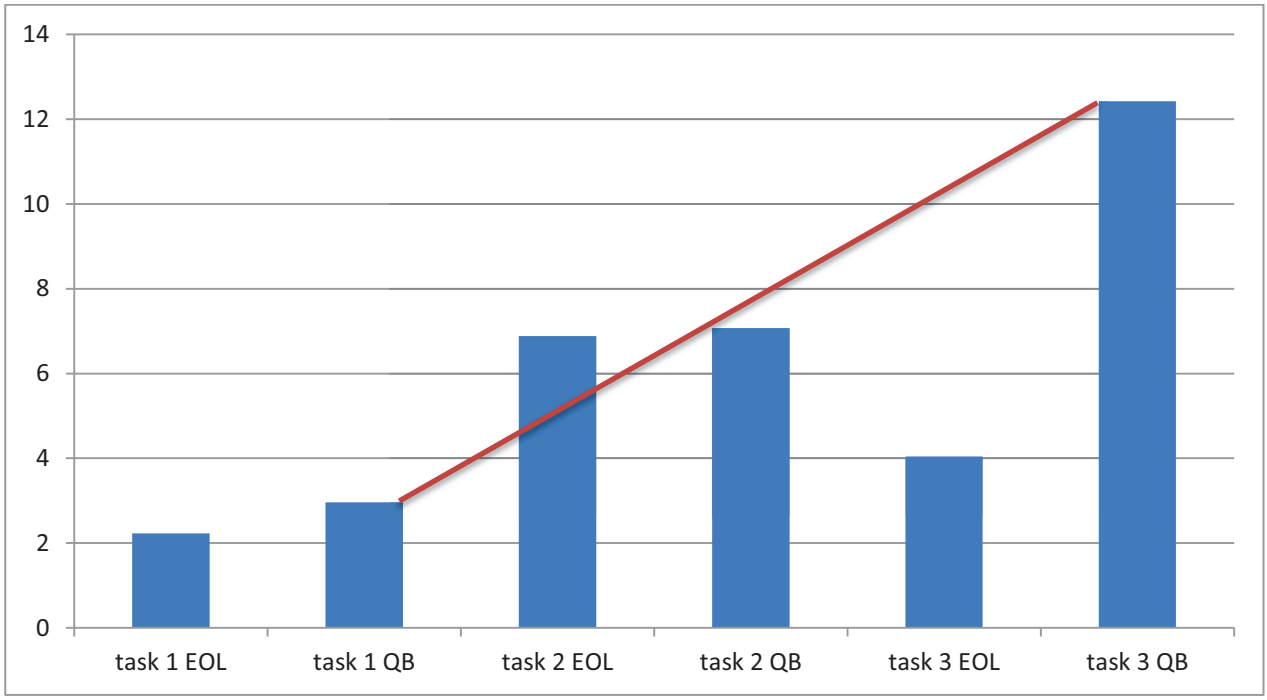

Figure 36 – Average time on task

6.3.1. GUI Troubles

The small difference in the average time on task between EOL and QB for task 2 can be explained by a GUI related problem that nearly all participants encountered with EOL. The second task involved the insertion of a grouping on sales person, which is done in the customization menu (depicted in figure 32). The participants needed quite some time to find the button (shown on the right side of the screen in figure 31) that opens the customization menu. In addition, the customization menu did not appear to be designed for typical end-users. Participants required much time and help before they were able to locate the group by section, to locate the sales person column, and to find out how to add this column to the group by section. This also explains why the number of help markers was higher in EOL (see figure 37).

Figure 37 – Average help/error markers per task

The average maximum time between inputs (figure 38) and the average number of mouse clicks (figure 39) support our explanation for the small difference with the second task a well. The maximum time between inputs (in seconds) is higher for EOL than for QB, and when participants spend much time searching for specific functionalities, the time between inputs will typically increase. In addition, we see that the average number of mouse clicks is quite a bit higher in QB compared with EOL while the time required to complete task 2 is almost equal for EOL and QB. Since participants spent relatively much time searching *where* to click, the number of clicks relative to the time on task decreases.

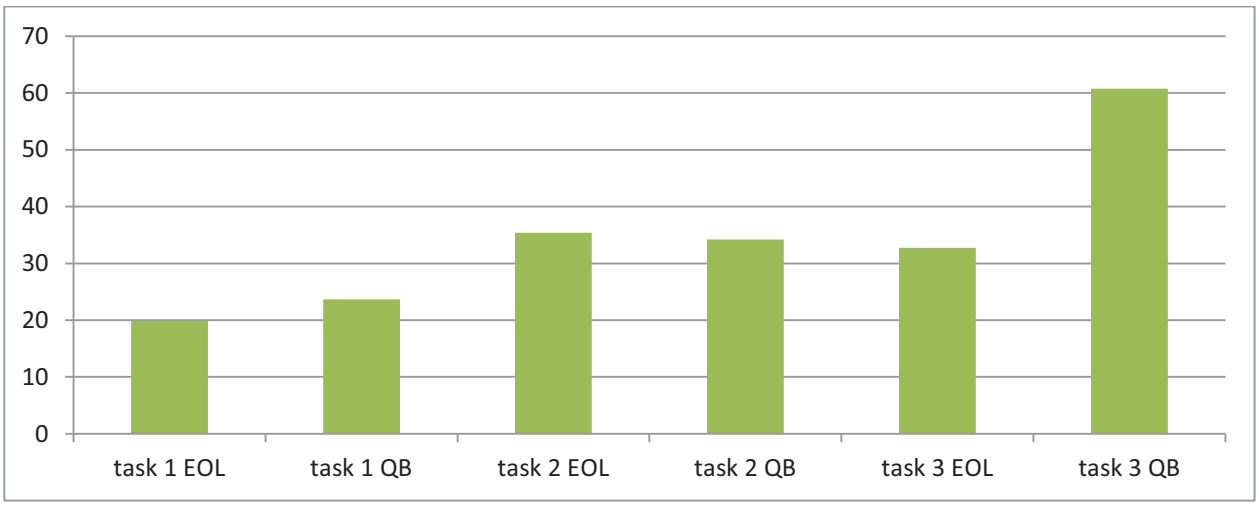

Figure 38 – Average maximum time between inputs

6.3.2. Flexibility and Task Complexity

Except for the second task, all usability metrics indicate a large difference between EOL and QB for the third task. In order to complete this task, participants had to create a graph showing revenue per month

+ year. Using the VP in EOL (shown in figure 35), participants had to select one of the predefined topics, filter the data to limit the report to a certain time span, select one of the predefined values that need to be shown in the graph, and choose how to group the data on predefined variables (e.g. year + period, GL account, year, et cetera). The most striking difference with the same task in the QB scenario is that participants had to create the year + period column and group the sales orders themselves, before the total order amount per year + period could be visualized in a graph. While the participants already knew how to group data as this was also involved in task 2, a typical error was that participants created a graph before grouping the data. This resulted in a graph that visualized all the order amounts of the orders issued during the selected period. Some participants did not forget to group the data, however, as they forgot to add a year + period column (while a tip on how to do this was included in the task description), participants grouped the sales orders per order date. These are good examples of errors that can be devoted to the task complexity caused by a high level of variability, rather than problems with the GUI. Considering the third task, the EOL VP was more limited than the QB VP as it was only possible to generate graphs showing totals (i.e. grouping was done automatically) and when the data was grouped it was only possible to show sums of values within this group (e.g. the total of the *sales order amount* of all sales orders per month, country, customer, et cetera). The QB VP thus provided flexibility in the way data was grouped (or not) and the way values were aggregated (e.g. the number of sales orders per country or the average sales order amount per period). While participants seemed conscious about the fact that the graph they first created in EOL showed summed order amounts, they forgot that they needed to perform this step themselves in the more flexible QB scenario.

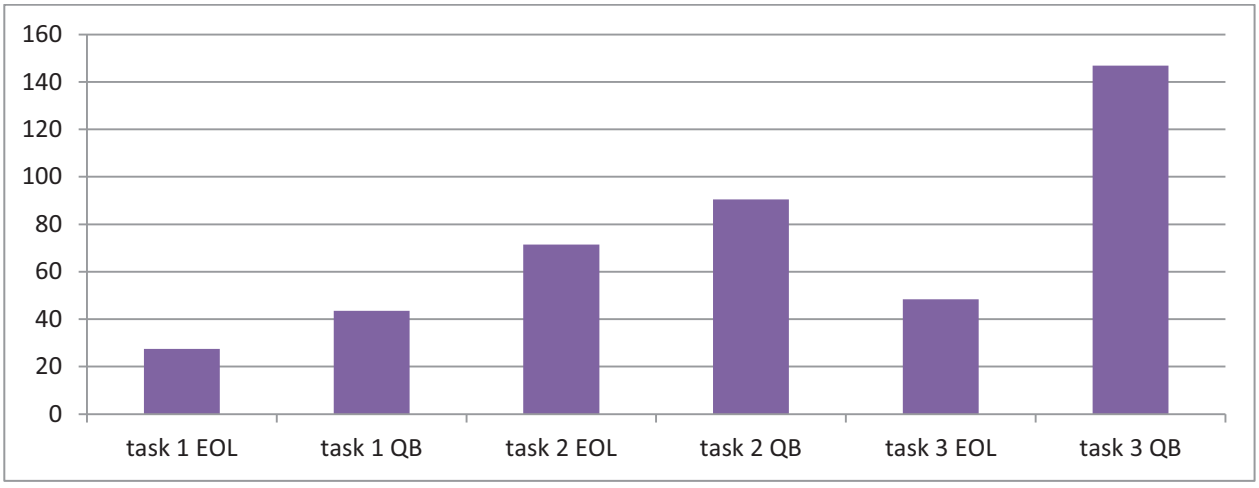

Figure 39 – Average # of mouse clicks

Summarizing we can state that for the third task, the large difference in task performance between EOL and QB can be devoted to the fact that in the QB scenario, participants had to perform more steps which not only resulted in more work (as can be derived from the metrics shown in figure 36 and 39), but above all resulted in a higher task complexity, which is supported by the metrics on the number of error and help markers and the average time between inputs (see figure 37 and 38). Participants made more errors and required more help in the QB scenario, indicating a decrease in task effectiveness when

flexibility increases. In addition, the relatively high maximum time between inputs for the QB VP can be explained by the increased task complexity caused by the increased flexibility.

6.3.3. Extended Learnability Influences Usability Metrics

Extended learnability refers to the change in task performance over time (Grossman et al., 2009). This contrasts initial learnability, which refers to the ability to perform well during an initial task. Due to the way we set up the experiment, each of the five usability metrics – the number of error and help markers and the time on task in particular – is influenced by extended learnability. In the first task we introduced concepts and functionalities which were also used in the second and third task. In addition, the first two tasks in EOL and all the three in QB have overlap in the functionalities used. When a participant performs a task for the second time, or uses functionality for the second time, participants will generally make less errors and require less help and time than they would make or require during the first performance. For example, this means that we expect that the task performance for task 2 (for both EOL and QB) would be worse when they did not perform task 1 first. When this experiment was designed in such a way that each participant would have performed only one task, the differences in task performance are expected to be higher. For example, when we take a look at the average error and help markers shown in figure 37, we see that there is only a very small difference between task 2 QB and task 3 QB, while the third tasks is considered to be more complex as completing the task requires the participant to combine a larger set of functionalities than in the second task. However, this influence on the metrics is *not* considered to be a problem as the main goal of this experiment is to compare the usability (i.e. initial task performance) of low flexibility VPs with that of high flexibility VPs. In other words, we are mainly interested in the difference in task performance between VPs used to perform the same task, rather than the influenced difference in task performance between tasks. Yet, in some cases – as is illustrated in the next section – it is important to be aware of this influence when drawing conclusions based on this experiment.

6.3.4. Usability versus Flexibility

As is depicted in figure 40, the usability metrics we have defined in the theoretical background can be categorized into task efficiency and task effectiveness. Efficiency, effectiveness and learnability are the sub factors we used in the operationalization of usability. However, as learnability can be measured by combining the efficiency and effectiveness metrics (see section 3.5) it is omitted from this analysis. Distinguishing between efficiency and effectiveness metrics enables more focused and thorough – bottom up – analysis on the relation between flexibility and usability.

6.3.4.1. Task Efficiency

One of the inevitable consequences of flexibility is that participants have to perform more actions to achieve the same results. For example, when a VP provides multiple ways in which values can be summarized (e.g. average, sum, etc.), the participants do not only have to specify on which column they want to group, but also how values need to be summarized. In contrast, when a VP supports only one

aggregation function, there is no need to specify what function should be applied – which decreases the workload. Since software providing more flexibility requires end-users to perform more actions, more flexibility means more work. When we take a look at the time on task metric (figure 36), we see that for each of the three tasks, the high flexibility scenario took more time. Although the difference between the low and high flexibility VP for task 2 is not significant in any way, the number of mouse clicks metric reveals that there were much more clicks required to complete the same task in a VP that provided more freedom. In addition, the metric on the average max time between inputs gives us an explanation for the small difference in the time that was required by participants to complete the second task. The low flexibility VP had a higher time between inputs than the high flexibility VP which implies that participants spent more time being unproductive in the low flexibility VP (e.g. searching the GUI for specific functionalities). In other words, from what can be observed from the usability metrics, completing task 2 in the low flexibility scenario was less work than in the high flexibility scenario. In summary, when increased the flexibility, the task efficiency degraded.

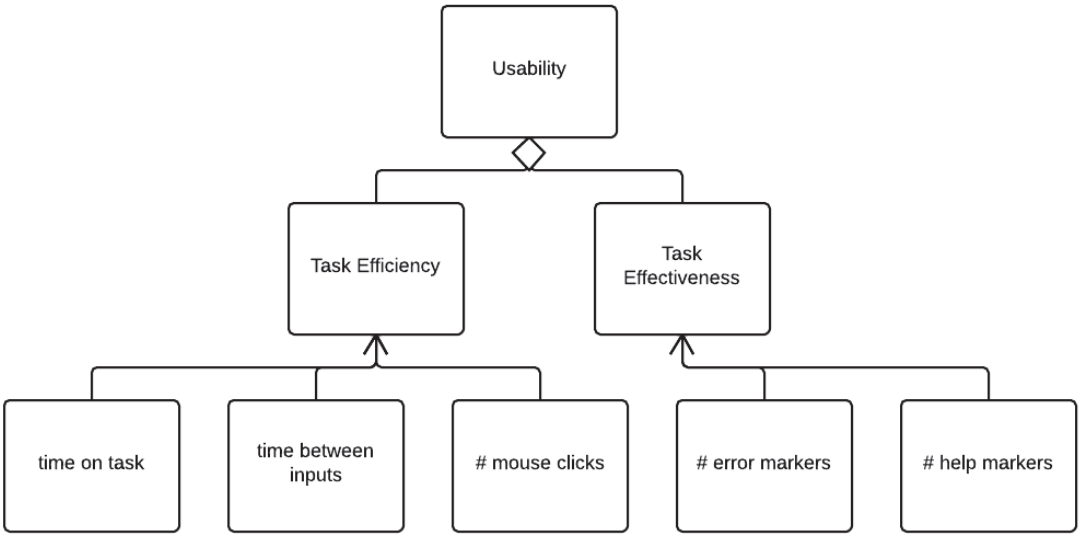

Figure 40 – Usability metrics categorized into task efficiency and task effectiveness

6.3.4.2. Task Effectiveness

When we take a look at figure 37 we see that participants made more errors and required more help when performing tasks using VPs providing a relatively high flexibility than when using VPs providing a relatively low flexibility. For all three tasks, more errors were made in the high flexibility (QB) scenario. This does not apply for the help metric, which shows us that for task 2, participants required more help in EOL (low flexibility). However, when we merge the help and error markers into one metric (as we did in figure 37), we can observe – as we did for the task efficiency – that the task effectiveness was lower in the high flexibility scenarios for all three tasks. This brings us to the conclusion that a negative relationship between usability and flexibility can be observed from the results of the usability evaluation.

6.3.5. Task Efficiency, Task Comprehensiveness and Flexibility

As mentioned in section 6.2.4, we deliberately increased the amount of required functionalities each task. The first task introduced filters, the second introduced grouping per column whereas the third task introduced the use of a chart to visualize all the previous. For the sake of conciseness, we refer to this as an increase in task comprehensiveness. In figure 36, a red line is drawn between the bars of task 1 QB and task 3 QB. The fact that the bar of task 2 QB is only barely below this line, indicates a nearly linear increase in the time on task in the high flexibility scenarios (QB). In other words, it can be observed from figure 36, that the time on task for QB increases when the task comprehensiveness is increased. In contrast, the time on task of the EOL does not show such relation at all. While we cannot really draw any conclusions based on this observation, it can lead to the presumption that an increase in task comprehensiveness causes a higher increase in the time on task for VPs providing a relatively high flexibility than for VPs providing a relatively low flexibility. The metrics on the average number of help and error marks (figure 37) do not show this linear relationship, as the difference between task 2 and task 3 (for QB) is too small. However, as explained earlier, the extended learnability – which is involved due the way this experiment was set up – might be the reason that the difference in initial task performance is too small to support the presumption. Considering the fact that we cannot prove the impact of the extended learnability, we speak merely of a possible triangular relationship between the task efficiency (rather than usability), flexibility and task comprehensiveness (figure 41).

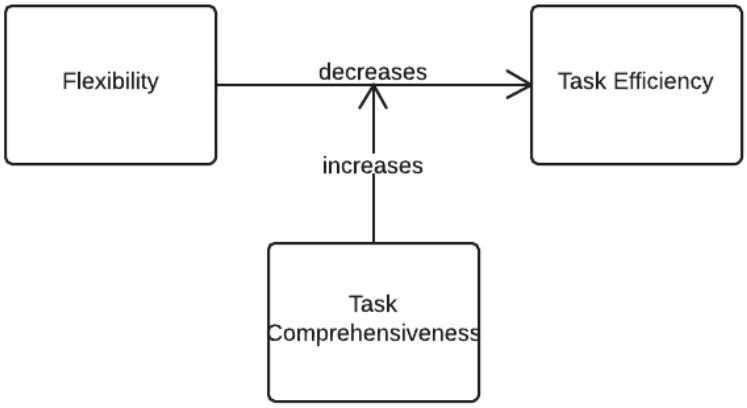

Figure 41 – Task comprehensiveness impacts relationship between flexibility and task efficiency

6.3.6. Questionnaire Results

Participants were asked to indicate their experience with Google Drive Spreadsheets, Microsoft Excel and business software (information systems) in general, on a scale from 1 to 10. Scores were 1.75 for Spreadsheets, 7.25 for Excel and 6.63 for business software. All participants answered with no to the question whether they had experience with Exact Online. Considering we compared the task performance between the EOL scenario and QB scenario, it makes sense to check the pre-experience participants had with the actual tools used in the usability experiment and tools that work in a similar ways using similar concepts. From the questionnaire results we can conclude that participants had no

experience with the EOL GUI and a very low experience using the Spreadsheets GUI. However, participants had some experience with software similar to both EOL and spreadsheets. In both the actual tools used during the experiment and tools similar to those tools, participants had more experience with the high flexibility (QB) scenario. For this reason, the system experience of the participant is not considered as a factor that influenced the results of this experiment.
Participants were also asked to provide an indication – on a scale from 1 to 10 – about how they felt about the GUIs of EOL (5.75) and QB (6.25) + Spreadsheets (5.75). As the subjective measures on the GUIs in the EOL and QB scenario do not indicate a large difference, we cannot conclude that the GUI had a serious influence on the task performance based on the questionnaire.
At last, all participants were higher educated (50% HBO and 50% WO) and indicated that they had sufficient financial knowledge in order to complete the tasks of the experiment.

6.4. Consequences for SaaS Vendors

The usability evaluation – performed to compare the usability of VPs providing relatively low degree flexibility to that of VPs providing a relatively high degree of flexibility – supports the presumed tradeoff between flexibility and usability. As is inherent to any tradeoff, the challenge is to find the proper balance between the two variables. This challenge is the concern of SaaS vendors developing multi-tenant applications.

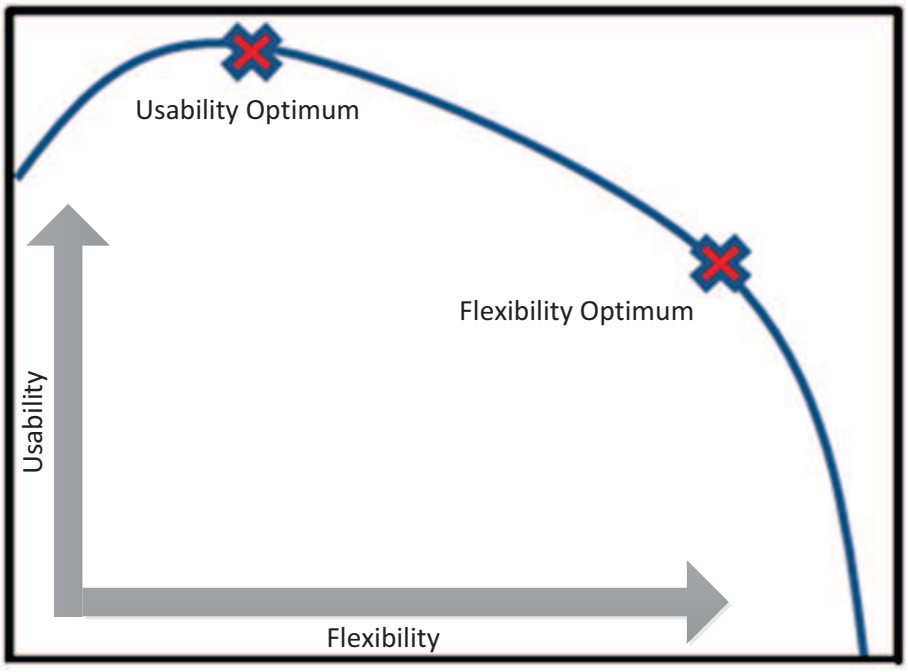

Figure 42 – Hypothesized curve of the relationship between flexibility and usability

The flexibility resulting from the implementation of software variability is advantageous for tenants as this enables them to adapt the software product to make it fit within their context. However, one of the pitfalls when implementing variability is that the usability – another quality attribute that should be maximized – can be seriously compromised. One might even argue that flexibility, which is even included

as a usability criteria in the model of Seffah et al. (2006), is a sub factor of usability. This would mean that when you start to increase flexibility, the perceived usability will increase as well. However, at a certain point – which totally depends on the context of use – the perceived usability starts to degrade as the increased flexibility introduces complexities that increasingly surpass the capacities of the end-user.
Based on this scenario, the abstract graph in figure 42 shows the relative degree of flexibility on the horizontal axis and the relative usability on the vertical axis. The blue line depicts the presumed curve of the tradeoff between usability and flexibility. The point at which flexibility is increased without negatively impacting usability is called the usability focused optimum. The flexibility focused optimum is the point at which flexibility is increased without degrading usability such that the software becomes unusable for its intended users. In other words, at the flexibility optimum, a higher degree of variability is implemented to support a larger range of tenant-specific requirements. With this approach, the usability of the product is compromised, though intended users are still able to use the software in an effective way. To summarize, the rationale behind these two optimums is based on two presumptions. The first is that the negative relationship between usability and flexibility is non-linear, as the impact of flexibility on the usability is amplified when flexibility increases. The second presumption is that a small increase in flexibility has a positive influence on the usability as perceived by end-users. This presumption is supported by the fact that in literature, flexibility is considered to be a sub factor of usability (Seffah et al., 2006).

6.4.1. The best of Both Sides

Assuming that the typical curve of the relationship between flexibility and usability is similar to the hypothesized curve shown in figure 42, software product vendors implementing variability in their multi-tenant environments should increase the flexibility of their product until they reach the usability focused optimum. A small portion of the (more advanced) users will also benefit from the flexibility within the area between the usability- and flexibility focused optimum. To serve this part of the customer base, this flexibility can be implemented on the condition that it is hidden from the basic users. We also observed this concept in the reporting framework of EOL (used for task 1) where the basic functionalities (figure 31) are separated from the advanced customization shown in figure 32 (used in task 2). This technique is an example of an approach that is aimed at implementing as much flexibility as possible without compromising the usability. In order to discover the positions of both the usability- and flexibility focused optimum, SaaS vendors can perform usability tests similar to the ones conducted in this research.
Other techniques to increase the usability by hiding redundant information and functionalities are described by the *Tenant-dependent View* and *Customizable Authorization Model* patterns described in chapter 5.

6.4.2. Context of Use

Before SaaS vendors can obtain reliable results from usability tests – which should be conducted in order to find out where to position the markers as shown in figure 42 – they should define a context of use. The ISO 9241-11 standard defined a comprehensive set of context of use attributes. SaaS vendors can select the attributes that are relevant for their software product and subsequently define the context of

use for these attributes (see chapter 3 for details on the context of use). For example, this research mainly focused on the user characteristics (one of the *context of use* attributes). The defined context of use resulted in participant selection criteria for the usability evaluations. These criteria were important as the user characteristics determine the degree of variability that can be implemented. For example, a variability point facilitating data model extension (i.e. facilitating the definition of custom business entities on run-time) can be way too complex to use for the typical user of EOL while users with knowledge on data modeling are perfectly able to operate this VP in a proper way. Attributes that we did not define, but that can be relevant in other situations include the characteristics of the task, physical environment or the social environment. For example, when a task is performed fifty times a day, the task efficiency becomes much more important whereas the social environment should be considered when the system was designed for a use context in which a high degree of assistance (e.g. helpful colleagues) is available.
Once the context of use have been defined, SaaS providers are enabled to perform usability experiments – similar to the one conducted during this study – to determine the extent to which flexibility can be implemented without seriously compromising the usability as perceived by the intended end-users.

6.5. Variability Implementation Approach

A high level variability implementation approach is defined in order to provide SaaS vendors with guidance on how the findings and experiences of this study can help them with the effective implementation of variability in multi-tenant environments (figure 43).
The implementation of variability is only required when it largely increases the potential customer base of the software by satisfying a broader range of tenant requirements. Based on the customer requirements, the areas in which the software product should be made more variable are determined. The next step is to implement (a prototype of) the variability in the multi-tenant environment. Before the variability point is designed, it is recommended to consult literature on variability patterns – such as the pattern language for tenant-dependent reporting that resulted from this study – to find solutions that can be applied to solve the problem at hand.
Once a (prototype) variability point is implemented, a usability evaluation should be performed in order to find out to what extent the intended users of the application are able to complete tasks using the variability point. The usability evaluation should be prepared by defining the intended context of use. This context should be simulated during the evaluation. For most business software, the characteristics of the intended users are the primary context of use attributes that need to be defined. Based on this, the participants can be selected. To provide context for the tasks that participants need to perform, a scenario needs to be defined. In this study, the question-suggestion tool proved to be a successful evaluation protocol to collect qualitative data whereas the Morae usability analysis tool was used to collect the quantitative data on the usability metrics. Depending on the type of software and variability point, the usability measures used in this study are suitable in most other scenarios as well. It is recommended to add a survey to collect data on extraneous variables that might influence the usability measures (e.g. participant knowledge and experience, subjective measures on the GUI of the VP, et cetera). Each evaluation session requires a separate analysis in which the recordings are used to collect and process both the quantitative and qualitative data. For this study, the recording captured the screen,

the facial expressions of the participant, and the verbalized thoughts and conversations between coach and participant.

Once the individual sessions have been performed and analyzed, a report showing the average scores per task is created. Based on this overall analysis, SaaS vendors have knowledge about the extent to which the introduced flexibility impacted usability. When participants were able to complete the tasks in an effective and efficient way, the change can be safely implemented within the multi-tenant application. However, when the measured usability was below the usability- or even the flexibility focused optimum, SaaS vendors can decide that the variability point needs adjustments to ensure that the intended users are actually able to use the introduced flexibility to customize the software to make it fit within their tenant-specific context.

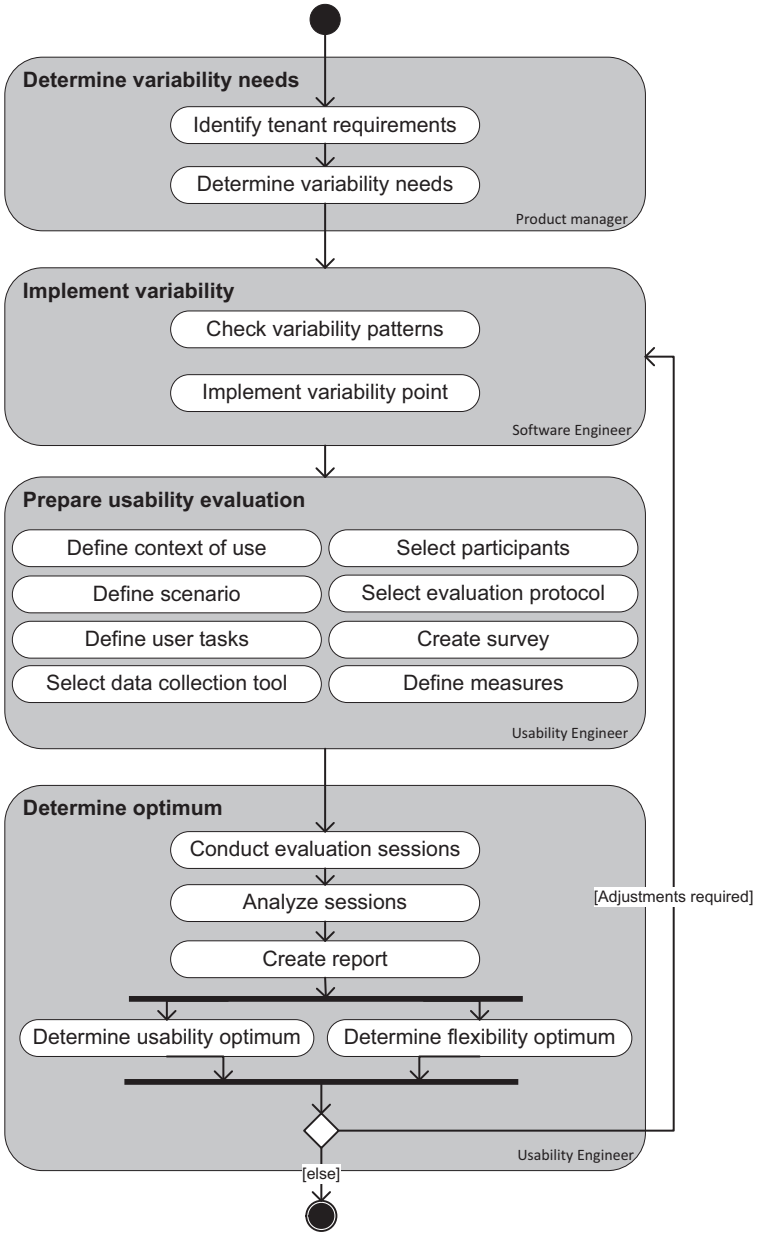

Figure 43 – The variability implementation approach

7 Conclusion

Multi-tenancy allows SaaS vendors to maximize their economies of scale by offering one shared application and databases instance to multiple customers. However, in order to satisfy as many customer requirements as possible, multi-tenant environments need to support tenant-based run-time variability as the shared nature of multi-tenancy makes it impossible to customize an application by changing its source code. Since customizing SaaS products is typically a task performed by end-users rather than consultants, usability is an important aspect that needs to be considered when implementing variability. Based on these business needs, the central research question was defined:

> *How can **tenant-dependent reporting** be implemented in **multi-tenant** environments and how do these different implementations impact the **usability** and **flexibility** provided to end-users?*

Reporting is one of the most important functional areas within business software. However, as no business is the same, requirements towards reporting are highly customer specific. In order to provide SaaS vendors with reusable solutions for the implementation of tenant-dependent reporting in multi-tenant environments, a pattern language is presented. A *Data Web API* can be implemented to empower tenants to extract their data out of the multi-tenant application. The *REST* pattern can be applied to increase the simplicity, visibility and scalability of the *Data Web API.* When enterprises use SaaS business software they should be able to perform their reporting activities using SaaS tools as well. SaaS vendors can implement *Data Middleware* with *Abstract Query Builder* to bridge the gap between various data sources – such as a *Data Web API* – and third party SaaS reporting tools.

During a case study on Exact Online (EOL), a prototype variability point was implemented, providing SaaS vendors with a detailed example on how the *Data Web API, REST, Data Middleware,* and *Abstract Query Builder* patterns can be applied to provide a way for tenants to select, filter, and sort their data from EOL, and import it into Google Drive Spreadsheets in which they can create their customized reports. Both a generic query builder web application and a data service within the RESTful OData web API on top of EOL were developed for this purpose.

Both the *Tenant-dependent View* and the *Customizable Authorization Model* pattern are combined with the *Reusable View Components* pattern describing a solution that SaaS vendors can apply in order to increase the usability of their multi-tenant application by enabling tenants to hide redundant information and functionalities.

A usability experiment was performed to gain insights into the hypothesized negative impact that the flexibility inherent to the implementation of variability has on the usability provided to end-users. Three user tasks were defined and for each task two variability points – that also served as input for the variability patterns – with a demonstrable difference in the degree of flexibility provided were selected. Results of the experiment indicate that usability – in both effectiveness and efficiency – is indeed negatively impacted by flexibility. SaaS vendors that want to extend their potential customer base by increasing the flexibility of their software should be aware of the adverse consequences for usability. In order to guide SaaS vendors in achieving an optimal balance between flexibility and usability, we propose a high level variability implementation approach.

8 Discussion and Future Research

The pattern language for tenant-dependent reporting contains reusable solutions which are abstracted from the reporting variability points that were identified during the case study at Exact. However, a large part of these reusable solutions are still candidate patterns, as their recurrence and goodness is not proved. More case studies are required to promote the candidate patterns into true patterns.

The patterns presented in this work all have an example section which describes the implementation from which it was abstracted. When applicable, implementations are described from a functional perspective, which is useful to illustrate the pattern´s consequences on flexibility and usability. For example, the *Tenant-dependent View, Customizable Authorization Model* and *Reusable view component* patterns describe solutions for the implementation of tenant-based variability enabling tenants to increase usability by removing non-required features from the GUI. However, there is a lack of information on the patterns' consequences for the usability of the customization process itself (i.e. the actions that lead to the customized view or authorization model). In addition, further research is required to collect more information on how the patterns can be applied effectively. Obviously, the variability patterns based on the prototype reporting variability point – i.e. the Query Builder web application and the RESTful API – are complemented with real world examples and experiences on the implementation of these reusable solutions in the context of EOL. However, the variability patterns based on variability points observed within EOL lack this kind of valuable information.

Considering the lack of research on the implementation of run-time variability and the emergence of multi-tenant software products, the industrial practice will also benefit from pattern languages describing proven solutions to variability problems other than those related to tenant-dependent reporting. Challenging problem areas include data model extension, workflow customization, and other meta-data driven architectures. Case studies on PaaS' (e.g. Force.com, Windows Azure, or Google App Engine) would generate interesting patterns for advanced software variability.

In order to test the extent to which the level of variability affects the usability provided by variability points, we have operationalized usability – the dependent variable of the usability experiment. However, this is contrasting to our approach towards the independent variable (i.e. flexibility). Although we have compared the usability of variability points having a demonstrably difference in the degree of flexibility provided to the end-user, further research is required to develop a more formal approach to describe and simulate this increase in flexibility. A first step towards this purpose is to define the concept of flexibility. Despite the fact that flexibility is hard to operationalize, it is possible to compare variability points offering similar functionality on the flexibility they provide, as was done with the EOL and QB VPs in this study. Future research is required to improve and formalize this relative classification.

The usability experiment was set up such that we had three opportunities to compare two variability points with a demonstrable difference in the flexibility they provided to end-users. Despite the fact that from all three cases (i.e. tasks) could be observed that the usability was lower for variability points providing higher flexibility, the difference between high and low flexibility VPs differed substantially among the different tasks. For the second task for example, we can hardly argue that the difference between the two VPs is significant, even though we have sufficient qualitative data to exemplify this small difference shown by the usability metrics. In this case, the GUI was an important factor affecting the results on the usability measures. This is one of the disadvantages of conducting such an experiment

using functionalities of existing software products. One of the advantages though, is that it is simpler to simulate a real life usage environment when using existing (successful) software products. By conducting the usability evaluation sessions according to the question-suggestion protocol, validity threats inherent to the VPs (e.g. when the GUI affects the usability measures) can be counteracted as this protocol facilitates the collection of qualitative data which can provide valuable context to the quantitative metrics.

Another factor influencing the results of the experiment is the selection of participants. What happens when participants are under- or overqualified? One can expect that in both directions, differences between VPs become smaller as participants that are overqualified might conduct the task flawless in both the low and high flexibility scenario, and the other way around. The fact that we used an existing software product for the experiment simplified the participant selection, as we could base the criteria on the target group defined by the case company (i.e. the software vendor). Considering the influence that factors like participant selection (and other context of use attributes) and VP characteristics (e.g. the GUI) have on the usability measures, more experiments should be conducted to confirm the outcomes of this study.

Besides an observed negative relationship between flexibility and usability, the usability experiment resulted in the presumption that this relationship is amplified by task comprehensiveness. More research is required to gain insights on whether and how task comprehensiveness impacts the relationship between flexibility and usability. Another presumption that needs further research is the shape of the curve of the relationship between flexibility and usability. Based on the usability evaluation sessions, we presume that the decrease in usability due to an increase of flexibility is not linear. Instead, we expect this effect to be amplified as the flexibility increases.

9 Acknowledgements

First of all, I would like to thank Jaap Kabbedijk for all the inspiring discussions we had and for guiding me throughout the entire research. I would also like to thank Slinger Jansen for acting as my second supervisor. I want to thank Vijai Ramcharan for his support, creativity and technical input and Edgar Wieringa for facilitating my research at Exact. Peter Dijst, Raymond Muiselaar, Peter van Katwijk, André van de Graaf, Wilko Frieke, Jenneke Taal, and many other people at Exact helped me collecting case study data. I also want to thank Onno Dijkstra for reviewing my work. At last, I would like to thank the financial staff of Plus Retail B.V. and all other people that participated in the usability experiment.

10 Bibliography

Alexander, C., Ishikawa, S., & Silverstein, M. (1977). *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press.

Anderson, C. (2004). *The long tail*. Business Books.

Antoniol, G., Fiutem, R., & Cristoforetti, L. (1998). Design pattern recovery in object-oriented software. In *Program Comprehension, 1998. IWPC'98. Proceedings., 6th International Workshop on* (pp. 153–160).

Apache CouchDB. (2013). Retrieved May 8, 2013, from http://couchdb.apache.org/

Bachmann, F., & Bass, L. (2001). Managing variability in software architectures. *ACM SIGSOFT Software Engineering Notes*, *26*(3), 126–132.

Bachmann, F., Bass, L., Carriere, J., Clements, P. C., Garlan, D., Ivers, J., … Little, R. (2000). Software architecture documentation in practice: Documenting architectural layers.

Beisiegel, M., Blohm, H., Booz, D., Edwards, M., Hurley, O., Ielceanu, S., … Marino, J. (2007). Sca service component architecture-assembly model specification.

Bezemer, C. P., & Zaidman, A. (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)* (pp. 88–92).

Bezemer, C. P., Zaidman, A., Platzbeecker, B., Hurkmans, T., & 't Hart, A. (2010). Enabling multi-tenancy: An industrial experience report. In *Software Maintenance (ICSM), 2010 IEEE International Conference on* (pp. 1 –8).

Bosch, J. (1999). Evolution and composition of reusable assets in product-line architectures: A case study. *Software architecture*, 321–339.

Brown, W. H., Malveau, R. C., & Mowbray, T. J. (1998). AntiPatterns: refactoring software, architectures, and projects in crisis.

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

Chong, F., & Carraro, G. (2006). Architecture strategies for catching the long tail. *MSDN Library, Microsoft Corporation*, 9–10.

Coplien, J., Hoffman, D., & Weiss, D. (1998). Commonality and variability in software engineering. *Software, IEEE*, *15*(6), 37–45.

Coplien, & Schmidt. (1995). *Pattern languages of program design* (Vol. 58). Addison-Wesley Harlow,, UK.

Crockford, D. (2006). The application/json media type for javascript object notation (json).

D'souza, A., Kabbedijk, J., Seo, D., Jansen, S., & Brinkkemper, S. (2012). Software-As-A-Service: Implications For Business And Technology In Product Software Companies. *PACIS 2012 Proceedings*.

Dropbox - Core API - REST reference. (2013). Retrieved May 8, 2013, from https://www.dropbox.com/developers/core/docs

Evelson, B. (2008). Topic overview: business intelligence. *Forrester Res November*, *21*.

Evelson, B. (2010). Want to know what Forrester's lead data analysts are thinking about BI and the data domain? | Forrester Blogs. Retrieved April 8, 2013, from

Fielding, R. (2000). *Fielding Dissertation: Chapter 5: Representational State Transfer (REST)*.

Fowler, M. (1997). *Analysis Patterns: Reusable Object Models*. Addison-Wesley Professional.

Fowler, M. (2003). *Patterns of enterprise application architecture*. Addison-Wesley Professional.

Fowler, M. (2004). *Inversion of control containers and the dependency injection pattern*.

Gamma, Helm, Johnson, & Vlissides. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software* (1st ed.). Addison-Wesley Professional.

Garlan, D. (2000). Software architecture: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering* (pp. 91–101).

Garrett, J. J. (2005). *Ajax: A new approach to web applications*.

Gomaa, H., & Hussein, M. (2007). Model-based software design and adaptation. In *Proceedings of the 2007 International Workshop on Software Engineering for Adaptive and Self-Managing Systems* (p. 7).

Grossman, T., Fitzmaurice, G., & Attar, R. (2009). A survey of software learnability: metrics, methodologies and guidelines. In *Proceedings of the 27th international conference on Human factors in computing systems* (pp. 649–658).

Guo, C. J., Sun, W., Jiang, Z. B., Huang, Y., Gao, B., & Wang, Z. H. (2011). Study of Software as a Service Support Platform for Small and Medium Businesses. *New Frontiers in Information and Software as Services*, 1–30.

Guo, Chang Jie, Sun, W., Huang, Y., Wang, Z. H., & Gao, B. (2007). A Framework for Native Multi-Tenancy Application Development and Management. In *E-Commerce Technology and the 4th IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services, 2007. CEC/EEE 2007. The 9th IEEE International Conference on* (pp. 551 –558).

Hardt, D. (2012). The OAuth 2.0 Authorization Framework.

Hevner, & Chatterjee, S. (2010). Design science research in information systems. *Design Research in Information Systems*, 9–22.

Hevner, March, S. T., Park, J., & Ram, S. (2004). Design science in information systems research. *MIS quarterly*, *28*(1), 75–105.

hOra, B. de, & Gregorio, J. (2007). The Atom Publishing Protocol.

Jacobs, D., & Aulbach, S. (2007). Ruminations on multi-tenant databases. *BTW Proceedings*, *103*, 514–521.

Jacobson, I., Griss, M., & Jonsson, P. (1997). Software reuse: architecture, process and organization for business success.

Jansen, S., Houben, G. J., & Brinkkemper, S. (2010). Customization realization in multi-tenant web applications: case studies from the library sector. *Web Engineering*, 445–459.

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

jQuery. (2013). Retrieved May 14, 2013, from http://jquery.com/

jQuery UI. (2013). Retrieved May 14, 2013, from http://jqueryui.com/

Kabbedijk, J., & Jansen, S. (2011). Variability in multi-tenant environments: architectural design patterns from industry. *Advances in Conceptual Modeling. Recent Developments and New Directions*, 151–160.

Kabbedijk, J., & Jansen, S. (2012). The role of variability patterns in multi-tenant business software. In *Proceedings of the WICSA/ECSA 2012 Companion Volume* (pp. 143–146).

Kang, S., Kang, S., & Hur, S. (2011). A Design of the Conceptual Architecture for a Multitenant SaaS Application Platform. In *2011 First ACIS/JNU International Conference on Computers, Networks, Systems and Industrial Engineering (CNSI)* (pp. 462 –467). Presented at the 2011 First ACIS/JNU International Conference on Computers, Networks, Systems and Industrial Engineering (CNSI). doi:10.1109/CNSI.2011.56

Kaplan. (2005). SaaS Survey Shows New Model Becoming Mainstream. *Cutter Consortium Executive Update*, *6*(22), 1–5.

Kaplan. (2007). Saas: Friend or foe? *Business Communications Review*, *37*(6), 48.

Keepence, B., & Mannion, M. (1999). Using patterns to model variability in product families. *Software, IEEE*, *16*(4), 102–108.

Kobielus, J., Karel, R., Evelson, B., & Coit, C. (2009). Mighty mashups: do-it-yourself business intelligence for the new economy. *Forrester Research*.

Krasner, G. E., & Pope, S. T. (1988). A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Journal of object oriented programming*, *1*(3), 26–49.

Kwok, T., Nguyen, T., & Lam, L. (2008). A Software as a Service with Multi-tenancy Support for an Electronic Contract Management Application. In *IEEE International Conference on Services Computing, 2008. SCC '08* (Vol. 2, pp. 179 –186). Presented at the IEEE International Conference on Services Computing, 2008. SCC '08.

Lehmann, S., & Buxmann, P. (2009). Pricing strategies of software vendors. *Business & Information Systems Engineering*, *1*(6), 452–462.

Lewis, C. (1982). *Using the" thinking-aloud" method in cognitive interface design*. IBM TJ Watson Research Center.

Li, H., Shi, Y., & Li, Q. (2009). A Multi-granularity Customization Relationship Model for SaaS. In *International Conference on Web Information Systems and Mining, 2009. WISM 2009* (pp. 611 – 615). Presented at the International Conference on Web Information Systems and Mining, 2009. WISM 2009.

Li, Liu, Li, & Chen. (2008). SPIN: Service performance isolation infrastructure in multi-tenancy environment. *Service-Oriented Computing–ICSOC 2008*, 649–663.

LINQPad. (2013). Retrieved May 10, 2013, from http://www.linqpad.net/

Luhn, H. P. (1958). A business intelligence system. *IBM Journal of Research and Development*, *2*(4), 314–319.

Meszaros, G., & Doble, J. (1998). A pattern language for pattern writing. *Pattern languages of program design*, *3*, 529–574.

Metzger, A., Heymans, P., Pohl, K., Schobbens, P.-Y., & Saval, G. (2007). Disambiguating the Documentation of Variability in Software Product Lines: A Separation of Concerns, Formalization and Automated Analysis. In *Requirements Engineering Conference, 2007. RE '07. 15th IEEE*

International (pp. 243 –253). Presented at the Requirements Engineering Conference, 2007. RE '07. 15th IEEE International.

Metzger, Andreas, & Pohl, K. (2007). Variability Management in Software Product Line Engineering. In *Companion to the proceedings of the 29th International Conference on Software Engineering* (pp. 186–187). Washington, DC, USA: IEEE Computer Society. doi:10.1109/ICSECOMPANION.2007.83

Michelsen, C. D., Dominick, W. D., & Urban, J. E. (1980). A methodology for the objective evaluation of the user/system interfaces of the MADAM system using software engineering principles. In *Proceedings of the 18th annual Southeast regional conference* (pp. 103–109).

Mietzner, R. (2008). Using variability descriptors to describe customizable saas application templates.

Mietzner, R., Leymann, F., & Papazoglou, M. P. (2008). Defining composite configurable saas application packages using sca, variability descriptors and multi-tenancy patterns. In *Internet and Web Applications and Services, 2008. ICIW'08. Third International Conference on* (pp. 156–161).

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

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

Morae. (2013). Retrieved April 2, 2013, from http://www.techsmith.com/morae.html

Müller, J., Krüger, J., Enderlein, S., Helmich, M., & Zeier, A. (2009). Customizing enterprise software as a service applications: Back-end extension in a multi-tenancy environment. *Enterprise Information Systems*, 66–77.

Natis, Y. V., Knipp, E., Valdes, R., Cearley, D. W., & Sholler, D. (2009). Who's Who in Application Platforms for Cloud Computing: The Cloud Specialists. *Gartner Research*.

Nielsen, J., & Hackos, J. T. (1993). *Usability engineering* (Vol. 125184069). Academic press Boston.

Nitu, M. (2009). Configurability in SaaS (software as a service) applications. In *Proceedings of the 2nd India Software Engineering Conference, Pune, India*.

OData. (2013). Retrieved May 10, 2013, from http://www.odata.org/

Pohl, K., Bockle, G., & Van Der Linden, F. (2005). *Software product line engineering* (Vol. 10). Springer.

Power, D. J. (2007). A brief history of decision support systems, *4*.

Radatz, J., Geraci, A., & Katki, F. (1990). IEEE standard glossary of software engineering terminology. *IEEE Std*, *610121990*, 121990.

Robson, C. (2002). *Real world research: a resource for social scientists and practitioner-researchers* (Vol. 2). Blackwell Oxford.

Rohleder, C., Davis, S., & Günther, H. (2005). Software Customization With XML. *Issues in Information Systems VI (2)*.

Runeson, P., & Höst, M. (2009). Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, *14*(2), 131–164.

Seffah, A., Donyaee, M., Kline, R. B., & Padda, H. K. (2006). Usability measurement and metrics: A consolidated model. *Software Quality Journal*, *14*(2), 159–178.

Sinnema, M., & Deelstra, S. (2007). Classifying variability modeling techniques. *Information and Software Technology*, *49*(7), 717–739.

Stake, R. E. (1995). *The art of case study research*. Sage Publications, Incorporated.

Standardization, I. O. for. (1998). *ISO 9241-11: Ergonomic Requirements for Office Work with Visual Display Terminals (VDTs): Part 11: Guidance on Usability*.

Sun, W., Zhang, X., Guo, C. J., Sun, P., & Su, H. (2008). Software as a service: Configuration and customization perspectives. In *Congress on Services Part II, 2008. SERVICES-2. IEEE* (pp. 18–25).

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

Tableau Software. (2013). Retrieved May 10, 2013, from http://www.tableausoftware.com/

Tsai, Shao, & Li. (2010). Oic: Ontology-based intelligent customization framework for saas. In *Service-Oriented Computing and Applications (SOCA), 2010 IEEE International Conference on* (pp. 1–8).

Van de Weerd, I., & Brinkkemper, S. (2008). Meta-modeling for situational analysis and design methods. *Handbook of research on modern systems analysis and design technologies and applications*, *35*.

Van den Haak, M. J., de Jong, M. D. T., & Schellens, P. J. (2004). Employing think-aloud protocols and constructive interaction to test the usability of online library catalogues: a methodological comparison. *Interacting with computers*, *16*(6), 1153–1170.

Van Gurp, J., Bosch, J., & Svahnberg, M. (2001). On the notion of variability in software product lines. In *Working IEEE/IFIP Conference on Software Architecture, 2001. Proceedings* (pp. 45 –54). Presented at the Working IEEE/IFIP Conference on Software Architecture, 2001. Proceedings. doi:10.1109/WICSA.2001.948406

Van Kesteren, A. (2010). Cross-origin resource sharing. *W3C Working Draft WD-cors-20100727*.

Walraven, S., Truyen, E., & Joosen, W. (2011). A middleware layer for flexible and cost-efficient multi-tenant applications. *Middleware 2011*, 370–389.

Warfield, B. (2007). Multitenancy can have a 16: 1 cost advantage over single-tenant. *SmoothSpan Blog*.

Watson, H. J., & Wixom, B. H. (2007). The current state of business intelligence. *Computer*, *40*(9), 96–99.

Weissman, C. D., & Bobrowski, S. (2009). The design of the force. com multitenant internet application development platform. In *Proceedings of the 35th SIGMOD international conference on Management of data* (pp. 889–896).

Xu, L., & Brinkkemper, S. (2005). Concepts of product software: Paving the road for urgently needed research.

Yin, R. K. (2008). *Case study research: Design and methods* (Vol. 5). Sage Publications, Incorporated.

Zhang, Y., Wang, Z., Gao, B., Guo, C., Sun, W., & Li, X. (2010). An effective heuristic for on-line tenant placement problem in SaaS. In *Web Services (ICWS), 2010 IEEE International Conference on* (pp. 425–432).

11 Appendix

11.1. Questionnaire

Based on your overall experience with all the tasks, please answer the following questions.

1. Do you have experience using Google Drive Spreadsheets?
   No experience 1 2 3 4 5 6 7 8 9 10 Highly skilled

2. Do you have experience using Microsoft Excel?
   No experience 1 2 3 4 5 6 7 8 9 10 Highly Skilled

3. Do you have experience using business software (information systems)?
   None 1 2 3 4 5 6 7 8 9 10 Highly experienced

4. Have you worked with Exact Online before?
   ( ) yes
   ( ) no

5. What level of education is typically required to fulfill your function?
   ( ) MBO
   ( ) HBO
   ( ) WO
   ( ) None of the above

6. Did you have sufficient financial knowledge in order to complete the tasks?
   ( ) yes
   ( ) no

7. How do you feel about the user interface of Google Drive Spreadsheets? Did the graphical design of the software helped you completing your tasks? Were you able to find the functionalities you needed to complete the tasks?
   very bad 1 2 3 4 5 6 7 8 9 10 very good

8. How do you feel about the user interface of Exact Online?
   very bad 1 2 3 4 5 6 7 8 9 10 very good

9. How do you feel about the user interface of the Query Builder web application?
   very bad 1 2 3 4 5 6 7 8 9 10 very good

10. Do you have any other comments?

11.2. User Tasks

Name	Description	Instructions
task 1 EOL	Reporting Framework	1. Open the first Chrome tab for the reporting framework. 2. Create an overview of the sales orders from the first six months of 2011 with sales person 'Sjaak de Groot'. 3. When sorted descending (Z->A) on order amount, what order number is shown in the first row?
task 1 QB	Query Builder	1. Open the second Chrome tab for the Query Builder. 2. Continue with step 2 and 3 of the previous task.
task 2 EOL	Reporting Framework + Customization menu	1. Open the first Chrome tab for the reporting framework 2. Create an overview with the sales orders since 2011 which are issued by customer 'Terrie Sport'. Group the data by Sales Person. 3. What sales person has the highest total order amount?
task 2 QB	Query Builder + Google Drive Spreadsheets	1. Open the second Chrome tab for the Query Builder. 2. Continue with step 2 and 3 of the previous task. Tip: First select the required data, then export to Google Drive Spreadsheets and create a pivot table to group data.
task 3 EOL	Customize graph	1. Open Firefox for EOL 2. Create a graph showing the revenue and gross margin per period (1/tm 12) of 2012 and 2013. 3. In what period was the highest revenue achieved?
task 3 QB	Query Builder + Google Drive Spreadsheets	1. Open the second Chrome tab for the Query Builder. 2. Create a graph showing the revenue per period of 2010 and 2011. Make sure it starts with period 1 of 2010 ascending (A->Z) to period 12 of 2011. 3. In what year + period were the highest revenue achieved? Tips: 1. Use the Query Builder to select, sort, and filter the required data before exporting to Google drive spreadsheets. 2. In Google Drive Spreadsheets: a. You can use the CONCATENATE function to merge the value of cells (i.e. year and month) into one cell. b. Use date functions YEAR and MONTH to extract the year and month from the date column.

11.3. Research Paper

A paper elaborating on the extent to which the level of variability affects the usability provided by variability points, is attached on the following pages.

Variability in Multi-tenant Environments: Usability versus flexibility in tenant-dependent reporting

Ruben Mijwaart[1], Jaap Kabbedijk[1], Vijai Ramcharan[2], and Slinger Jansen[1]

[1] Utrecht University, Department of Information and Computing Sciences, P.O. Box 80.007, 3508 TA, Utrecht, the Netherlands
h.r.mijwaart@students.uu.nl
{j.kabbedijk,s.jansen}@uu.nl

[2] Exact International Development B.V., P.O.Box 5066, 2600 GB Delft, the Netherlands
vijai.ramcharan@exact.com

Abstract. Multi-tenant applications enable maximization of economies of scale by offering one shared application and database instance to multiple customers. However, as the shared nature of multi-tenancy makes it impossible to customize by changing source code, tenant-based runtime variability needs to be implemented to satisfy as many customer requirements as possible. Since tenants became responsible for customizing the software, usability is an important aspect to consider when implementing variability. However, the extent to which the level of variability affects usability is unknown. A usability experiment following the question-suggestion protocol was performed to test the hypothesized negative relationship between flexibility and usability. Results indicate that usability, in both effectiveness and efficiency, is indeed negatively impacted by flexibility. Software vendors that want to extend their potential customer base by increasing the flexibility of their software should be aware of the adverse consequences for usability as a serious decrease in usability causes customers to cancel their subscription. Considering the presumed non-linearity of the curve of the relationship between flexibility and usability, a flexibility- and usability focused optimum can be found using the usability evaluation approach of this study. Once these optima are found, a detailed variability implementation strategy can be defined.

1 Introduction

Satisfying the varying requirements of individual tenants by the design of variability within multi-tenant environments is a crucial aspect for SaaS vendors in order to attract a significant number of tenants [4,6,5,8]. The more requirements a software product supports, the bigger the market that can be served with a single application instance. This principle applies not only to multi-tenant SaaS products, but also for software products in general. However, in single-tenant

applications customization can also be done by simply changing the source code since there is only one tenant using this code-base. Software vendors (or their implementation partners) selling on-premise software typically offer this type of customization as an additional service. In multi-tenant environments, customizing by changing source code is not possible as this code base is shared amongst multiple customers. For this reason, multi-tenant SaaS requires run-time variability to make the application applicable in a broader context (i.e. supporting more customer requirements).

Software variability refers to the ability of a software system to be changed for use in a particular context [12]. Besides variability, also configurability and customizability are terms used to describe the ability of software to be changed to fit within a specific context. Configurability refers to the ability of a system to be changed within a pre-defined scope (e.g. through wizards and configuration settings) whereas customizability typically implies the ability of a system to be changed by adding custom code to extend the application with custom functionality. A point in the software facilitating delayed design decision is referred to as a variability point [12].

While customization of on-premise software is typically performed by the software vendor or implementation partners, the customization of multi-tenant applications – which are particularly interesting for SMEs due to the lower overall costs [1] – is often done by the customers themselves. This is advantageous for customers as they can avoid consultancy costs. However, this applies only when SaaS vendors succeed to make the customization process as simple as possible [2]. This paper addresses the presumed negative relationship between the flexibility – inherent to the implementation of variability – and the usability provided by variability points. This hypothetical tradeoff is also illustrated by the aforementioned concepts of configurability and customizability. From an end-user perspective, changing an application's behavior by using built-in configurability is much simpler than utilizing the customizability of an application. However, customizability provides much more freedom (i.e. flexibility) than the pre-defined scope implied by configurability. For example, configuring the UI such that it reflects corporate branding will generally be a simple task for end-users to complete, whereas on the other extreme, the run-time variability introduced by the APEX programming language requires skilled developers to implement customizations on the multi-tenant environment of Force.com [13].

Software vendors that want to extend their potential customer base by increasing the flexibility of their software should be aware of adverse consequences for usability. Supporting a wide range of customer requirements is important to increase sales. However, when the increased flexibility causes the product to be inoperable by the intended users, customers will cancel their subscriptions. For this reason, the following research question is addressed by this study:

To what extent does the level of variability affect the usability provided by variability points?

2 Research Approach

In order to test the presumed negative relationship between flexibility and usability, a usability experiment following the question-suggestion protocol was conducted. Eight participants performed three tasks in both a relatively limited variability point (VP) and a relatively flexible VP. During the usability study, data was collected on five measures to indicate the extent to which an increased flexibility negatively impacts usability. A case study on Exact Online (EOL), the multi-tenant application offered by Exact, was performed in order to identify reporting variability points (VP). Two of the VPs from EOL are used in the low flexibility scenario of the usability experiment. In addition, a more flexible prototype variability point is developed within the context of EOL which is used as the high flexibility scenario in the usability experiment. The next section elaborates on the case study and provides details about the developed prototype. Section 2.2 explains how the difference in flexibility amongst the VPs is determined. Section 2.3 describes the user tasks performed by the participants in both the high and low flexibility scenario. Section 2.4 explains why the question-suggestion protocol was used whereas section 2.5 concerns the operationalization of the concept of usability.

2.1 Case Study

Currently, more than 150 employees are dedicated to EOL and the product is available in the Netherlands, Belgium, UK, and USA. In total, the product has more than 26k tenants of which around 90% are companies with less than 20 employees and around 40% are sole proprietorships. In addition to the identification of existing VPs, a prototype reporting variability point was implemented within the context of EOL. The goal of this prototype is to test a new way in which tenants are provided with more reporting flexibility. For this purpose, the RESTful OData API of EOL was extended with a data service, enabling tenants to extract their sales order data. In addition, a query builder web application (QB) was developed which uses the API of EOL to extract data, and the API of Google Drive Spreadsheets to create new spreadsheets containing the data from EOL. The QB provides a GUI (similar to Microsoft Excel) that enables tenants to create an OData query by selecting a data topic and subsequently selecting and sorting columns and applying filters. The main advantage of the QB usage scenario (the QB web appliction in combination with Google Drive Spreadsheets) is that it provides a lightweight solution in the cloud. Tenants that use SaaS information systems might want to do SaaS reporting as well. Considering the small size of the typical EOL customer, more sophisticated reporting solutions might be too complex and too expensive.

2.2 Variability Points

Exact Online (EOL) contains various kinds of VPs which enable end-users to customize what data is represented and how this data is represented. However,

in most cases the VPs within EOL are rather limited in comparison with the flexibility provided by the QB usage scenario. For example, EOL provides functionality to group data, but only the SUM aggregation function can be used and grouping is only possible on pre-defined columns. Another limitation can be illustrated by the filter functionalities of EOL. Users can only select sales orders within one time span whereas QB enables users to compare the revenue of 2013 with that of 2011 by selecting sales orders from two time spans (i.e. applying multiple filters on the date column).

2.3 User tasks

Based on the VPs of EOL, three tasks have been defined. Participants performed each task twice, once with a less flexible VP of EOL and once in the more flexible QB. By measuring the task performance of the participants, there are three opportunities in which the impact of flexibility on usability can be observed. The objective during the first task was to create an overview of the sales orders of one specific sales person within a certain time span. During the second task, participants had to create an overview of sales orders issued by one specific customer within a certain time span grouped by sales person. The objective of the last task was to create a chart showing the total revenue per period (year + month) within a specific time span.

2.4 Question-suggestion Protocol

The *concurrent thinking-aloud protocol* is the predominant data collection method for usability evaluation [11]. With this approach, participants are verbalizing their thoughts while performing predefined tasks using the evaluated artifact. Other data collection methods include the *retrospective thinking-aloud*, which differs from the concurrent thinking-aloud in that the thoughts are verbalized after the tasks have been completed [11]. With the *constructive interaction method* (i.e. co-discovery), two participants work together on a task while verbalizing their thoughts. In the *question-suggestion protocol* – which is based on the *question-asking protocol* – participants can not only ask questions to the coach, but in addition, the coach is free to give advice [3]. The study of [11] compared the concurrent- and retrospective thinking-aloud and the constructive interaction protocols on their effectiveness for identifying usability problems. No significant differences in their effectiveness have been found. However, using the question-suggestion protocol proved to be more effective in identifying usability issues compared to the thinking-aloud protocol [3]. For this reason, the question-suggestion protocol is applied within this study.

2.5 The Operationalization of Usability

In order to be able to measure the usability provided by variability points, the concept we call usability needs to be defined and operationalized. Although this

study is not addressing the question of what usability really is, we need a clear definition of the usability that is truly inherent to the implementation of software variability. Two examples of high level definitions originating from well-known standards that largely correspond to our view of this type of usability are as follows:

- "*The extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use*" (ISO 9241-11, 1998)
- "*The ease with which a user can learn to operate, prepare inputs for, and interpret outputs of a system or component*" (IEEE Std.610.12, 1990)

Both of the above definitions do certainly not exclude the usability characteristics that we deem to be inherent to variability points. However, we require a more narrow definition merely including relevant aspects such that metrics can be defined. The work of Seffah et al. [10] describes a literature study across different usability standards and methods, resulting in a consolidated model in which ten usability sub factors are included. Learnability, effectiveness and efficiency are the usability sub factors constituting the dependent variable of this study.

2.6 Context of Use

As is also suggested by the usability definition of the ISO 9241-11 standard, one can only make a proper judgment on the usability of a software system when the context of use is defined [10]. The 'context of use attributes' defined in ISO 9241-11 include the characteristics of the user, task (e.g. frequency, duration, importance), technical environment (e.g. operating system, hardware capabilities and constraints), physical environment (e.g. noise level, privacy), organizational environment (e.g. structure of operational teams and the user's level of autonomy), and the social environment (e.g. degree of assistance available). For this study, the user characteristics are most applicable and those should thus be defined preparatory to the actual measurements.

User characteristics include task experience, application experience, system experience (e.g. knowledge of computer, OS and browser), and even psychological attributes like the cognitive style (e.g. motivation to use the system and analytic versus intuitive style). The task experience relates to the knowledge of the domain, whereas the application experience concerns the users' experience with similar applications. To illustrate the importance of defining the context of use before measuring usability, consider a variability point that is empowering users to build their own queries which can be used to generate reports. This specific functionality might be too complex for a typical bookkeeper without any knowledge of data persistence. A business intelligence professional on the other hand, won't have any problems with effectively using this functionality.

For this study, we selected participants having sufficient domain (i.e. financial) knowledge and system experience (e.g. familiar with browsers and spreadsheet software like Excel or Google Drive Spreadsheets) required to perform the

defined user tasks. However, participants should not be technical experts (e.g. no BI specialists etc.). In addition, participants should be unfamiliar with the evaluated variability points (i.e. no existing users of Exact Online) and should have sufficient knowledge of the English language, as this is used in the GUIs of the VPs.

2.7 Efficiency

Efficiency refers to the "*capability of a software product to enable users to expend appropriate amounts of resources in relation to the effectiveness achieved in a specified context of use*" [10] – another sub variable of usability. Efficiency can be considered as one of the measurable criteria of learnability. When tasks are getting more complex, the efficiency might decrease. For this reason, we consider efficiency as one of the aspects of usability that is related to the concept we attempt to measure.

2.8 Effectiveness

Another sub variable of usability which is considered relevant to this research is effectiveness, which is defined as "*the capability of a software product to enable users to achieve specified tasks with accuracy and completeness*" [10]. When an increased flexibility increases the task complexity, participants might be unable to complete the task without help or without making errors.

2.9 Learnability

Learnability is by far the most relevant sub factor related to the usability of a variability point. When one expects usability to decrease with an increase of flexibility, in a sense, usability is used as a synonym of simplicity. In other words, the use case complexity increases when the flexibility provided by the variability point increases. The sub factor reflecting this complexity is learnability. Based on the learnability definitions found across literature, one can distinguish between initial learnability and extended learnability [3]. Extended learnability refers to the change in task performance over time whereas initial learnability concerns *"the ability to perform well during an initial task for a user that is unfamiliar with similar functionality and whose domain knowledge is optimal"*. This study is focused on the initial learnability provided by variability points as this can be an indication for the presumed increase of complexity caused by an increased flexibility. In addition, initial learnability is easier to measure since extended learnability requires users to participate multiple times in usability tests spread over a certain time frame.

2.10 Metrics

In order to obtain a usability score for each of the evaluated variability points, five metrics are defined. Note that one and the same metric can be a measure for

multiple sub factors, while each sub factor can be measured by a combination of metrics. The main goal of the operationalization is that each sub factor is measured by at least one metric, and that all metrics (and thus all sub factors) together constitute a measure to gain insights in the usability concept defined for this research. The first three metrics are related to the efficiency whereas the fourth and fifth metrics address the effectiveness. Since both task performance and task effectiveness are sub factors of learnability [3], all defined metrics are indicators of initial learnability.

1. Time until a user completes a certain task successfully [9].
2. The number of mouse clicks required by participants to perform a certain task.
3. The maximum time in seconds between keystrokes or mouse clicks during a specific task.
4. The number of suggestions that a coach provided to a participant during a certain task.
5. The number of errors made by users performing a certain task [7]. An error is defined as any action that does not contribute to the desired result [9].

3 Results

The time on task (figure 1), error and help markers (figure 2), and number of mouse clicks metrics show that the flexible usage scenario resulted in a lower usability for each of the three tasks. However, differences between EOL and QB are very small in task two whereas all measures show a large difference for task three. During the second task, participants had problems to find and use the group by functionality. This explanation of the small difference is supported by the maximum time between inputs metric shown in figure 4. The time between inputs was higher for EOL than for QB which is an expected result when participants spent relatively much time searching the GUI for required functionalities. As the explanation that was based on the qualitative input of the usability sessions is supported by the metrics, we can conclude that the GUI was an extraneous variable that caused a decrease in the usability metrics for EOL.

For the third task, the large difference in task performance between EOL and QB can be devoted to the fact that in the QB scenario, participants had to perform more steps which not only resulted in more work (as can be derived from the metrics shown in figure 1 and 3), but above all resulted in a higher task complexity, which is supported by the metrics on the number of error and help marks and the average time between inputs (see figure 2 and 4). Participants made more errors and required more help in the QB scenario, indicating a decrease in task effectiveness when flexibility increases. In addition, the relatively high maximum time between inputs for the QB VP can be explained by the increased task complexity caused by the increased flexibility.

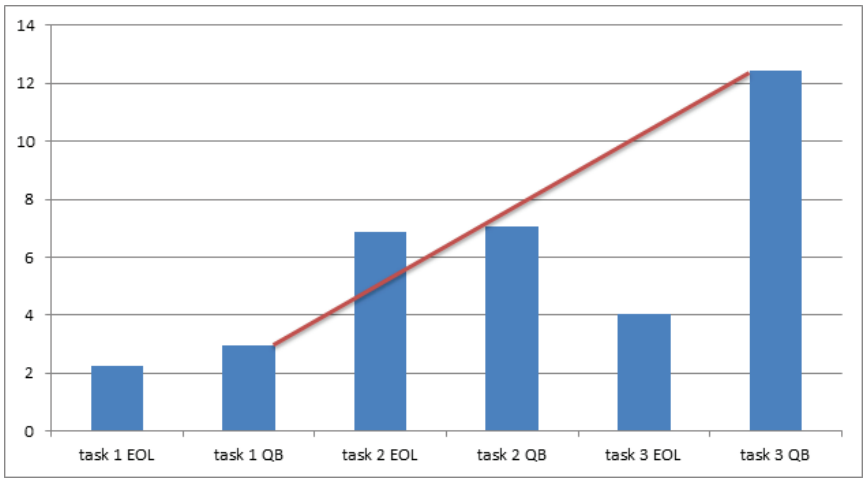

Fig. 1. Average time on task per task

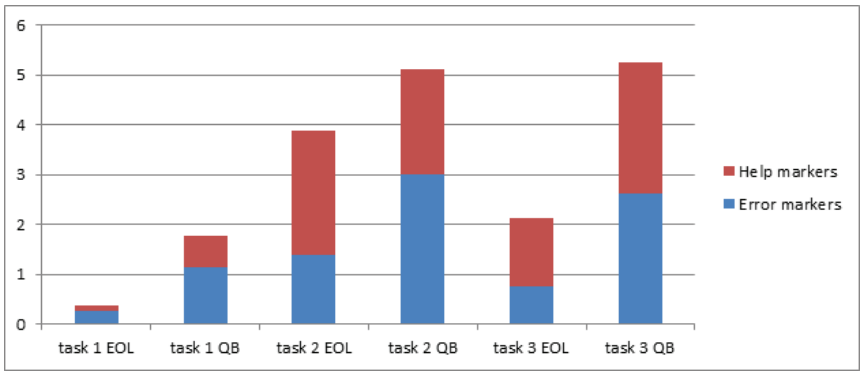

Fig. 2. Average number of help and error marks per task

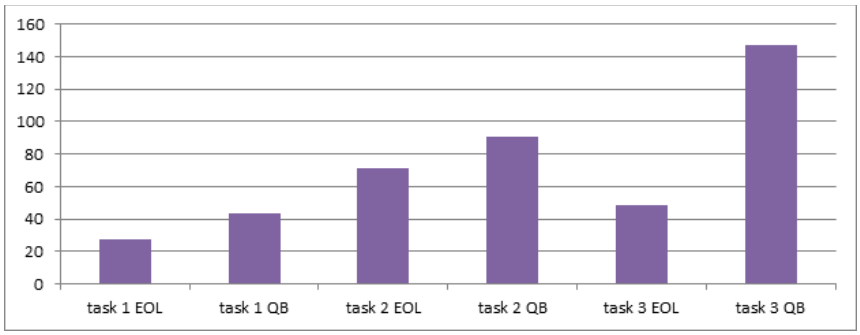

Fig. 3. Average number of clicks per task

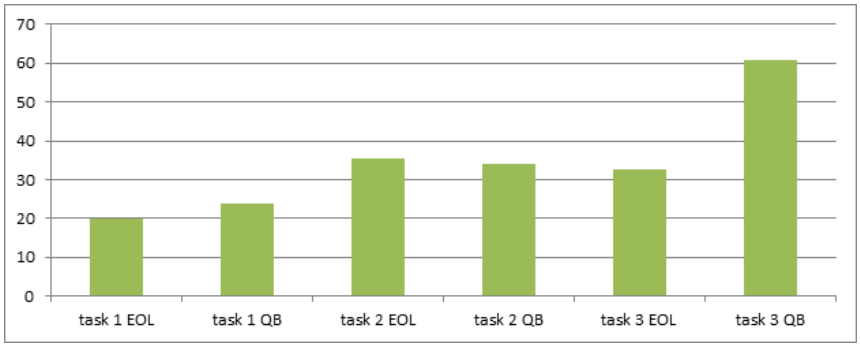

Fig. 4. Average time between inputs per task

3.1 Extended Learnability Influences Usability Metrics

Extended learnability refers to the change in task performance over time [3]. This contrasts initial learnability, which refers to the ability to perform well during an initial task. Due to the way we set up the experiment, each of the five usability metrics – the number of error and help markers and the time on task in particular – is influenced by extended learnability. In the first task we introduced concepts and functionalities which were also used in the second and third task. In addition, the first two tasks in EOL and all the three in QB have overlap in the functionalities used. When a participant performs a task for the second time, or uses functionality for the second time, participants will generally make less errors and require less help and time than they would make or require during the first performance. For example, this means that we expect that the task performance for task 2 (for both EOL and QB) would be worse when they did not perform task 1 first. When this experiment was designed in such a way that each participant would have performed only one task, the differences in task performance are expected to be higher. For example, when we take a look at the average error and help markers shown in figure 2, we see that there is only a very small difference between task 2 QB and task 3 QB, while the third tasks is considered to be more complex as completing the task requires the participant to combine a larger set of functionalities than in the second task. However, this influence on the metrics is not considered to be a problem as the main goal of this experiment is to compare the usability (i.e. initial task performance) of low flexibility VPs with that of high flexibility VPs. In other words, we are mainly interested in the difference in task performance between VPs used to perform the same task, rather than the influenced difference in task performance between tasks. Yet, in some cases – as is illustrated in the next section – it is important to be aware of this influence when drawing conclusions based on this experiment.

3.2 Task Efficiency versus Flexibility

One of the inevitable consequences of flexibility is that participants have to perform more actions to achieve the same results. For example, when a VP provides multiple ways in which values can be summarized (e.g. average, sum, etc.), the participants do not only have to specify on which column they want to group, but also how values need to be summarized. In contrast, when a VP supports only one aggregation function, there is no need to specify what function should be applied – which decreases the workload. Since software providing more flexibility requires end-users to perform more actions, more flexibility means more work. When we take a look at the time on task metric (figure 1), we see that for each of the three tasks, the high flexibility scenario took more time. Although the difference between the low and high flexibility VP for task 2 is not significant in any way, the number of mouse clicks metric reveals that there were much more clicks required to complete the same task in a VP that provided more freedom. In addition, the metric on the average max time between inputs gives us an explanation for the small difference in the time that was required by participants to complete the second task. The low flexibility VP had a higher time between inputs than the high flexibility VP which implies that participants spent more time being unproductive in the low flexibility VP (e.g. searching the GUI for specific functionalities). In other words, from what can be observed from the usability metrics, completing task 2 in the low flexibility scenario was less work than in the high flexibility scenario. In summary, when increased the flexibility, the task efficiency degraded.

3.3 Task Effectiveness versus Flexibility

When we take a look at figure 2 we see that participants made more errors and required more help when performing tasks using VPs providing a relatively high flexibility than when using VPs providing a relatively low flexibility. For all three tasks, more errors were made in the high flexibility (QB) scenario. This does not apply for the help metric, which shows us that for task 2, participants required more help in EOL (low flexibility). However, when we merge the help and error markers into one metric (as we did in figure 2), we can observe – as we did for the task efficiency – that the task effectiveness was lower in the high flexibility scenarios for all three tasks. This brings us to the conclusion that a negative correlation between usability and flexibility can be observed from the results of the usability evaluation.

3.4 Task Efficiency, Task Comprehensiveness and Flexibility

As mentioned in section 2.3, we deliberately increased the amount of required functionalities each task. The first task introduced filters, the second introduced grouping per column whereas the third task introduced the use of a chart to visualize all the previous. For the sake of conciseness, we refer to this as an increase in task comprehensiveness. In figure 1, a red line is drawn between the

bars of task 1 QB and task 3 QB. The fact that the bar of task 2 QB is only barely below this line, indicates a nearly linear increase in the time on task in the high flexibility scenarios (QB). In other words, it can be observed from figure 1, that the time on task for QB increases when the task comprehensiveness is increased. In contrast, the time on task of the EOL does not show such relation at all. While we cannot really draw any conclusions based on this observation, it can lead to the presumption that an increase in task comprehensiveness causes a higher increase in the time on task for VPs providing a relatively high flexibility than for VPs providing a relatively low flexibility. The metrics on the average number of help and error marks (figure 2) do not show this linear relationship, as the difference between task 2 and task 3 (for QB) is too small. However, as explained earlier, the extended learnability – which is involved due the way this experiment was set up – might be the reason that the difference in initial task performance is too small to support the presumption. Considering the fact that we cannot prove the impact of the extended learnability, we speak merely of a possible triangular relationship between the task efficiency (rather than usability), flexibility and task comprehensiveness (figure 5).

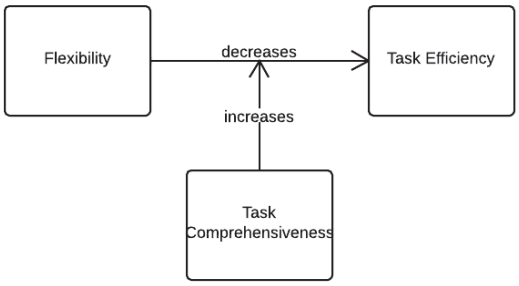

Fig. 5. task comprehensiveness impacts the relationship between flexibility and task efficiency

3.5 Consequences for SaaS Providers

The usability evaluation – performed to compare the usability of VPs providing relatively low degree flexibility to that of VPs providing a relatively high degree of flexibility – supports the presumed tradeoff between flexibility and usability. As is inherent to any tradeoff, the challenge is to find the proper balance between the two variables. This challenge is the concern of SaaS vendors developing multi-tenant applications. The flexibility resulting from the implementation of software variability is advantageous for tenants as this enables them to adapt the software product to make it fit within their context. However, one of the

pitfalls when implementing variability is that the usability – another quality attribute that should be maximized – can be seriously compromised. One might even argue that flexibility, which is even included as a usability criteria in the model of [10], is a sub factor of usability. This would mean that when you start to increase flexibility, the perceived usability will increase as well. However, at a certain point – which totally depends on the context of use – the perceived usability starts to degrade as the increased flexibility introduces complexities that increasingly surpass the capacities of the end-user. Based on this scenario, the abstract graph in figure 6 shows the relative degree of flexibility on the horizontal axis and the relative usability on the vertical axis. The blue line depicts the presumed curve of the tradeoff between usability and flexibility. The point at which flexibility is increased without negatively impacting usability is called the usability focused optimum. The flexibility focused optimum is the point at which flexibility is increased without degrading usability such that the software becomes unusable for its intended users. In other words, at the flexibility optimum, a higher degree of variability is implemented to support a larger range of tenant-specific requirements. With this approach, the usability of the product is compromised, though intended users are still able to use the software in an effective way. To summarize, the rationale behind these two optimums is based on two presumptions. The first is that the negative relationship between usability and flexibility is non-linear, as the impact of flexibility on the usability is amplified when flexibility increases. The second presumption is that a small increase in flexibility has a positive influence on the usability as perceived by end-users. This presumption is supported by the fact that in literature, flexibility is considered to be a sub factor of usability [10].

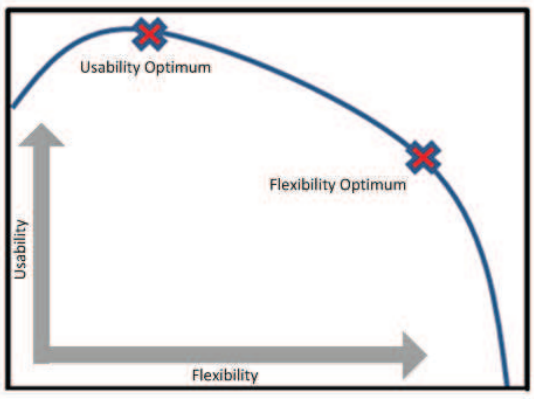

Fig. 6. hypothesized curve of the relationship between flexibility and usability

3.6 The best of Both Sides

Assuming that the typical curve of the relationship between flexibility and usability is similar to the hypothesized curve shown in figure 6, software product vendors implementing variability in their multi-tenant environments should increase the flexibility of their product until they reach the usability focused optimum. A small portion of the (more advanced) users will also benefit from the flexibility within the area between the usability- and flexibility focused optimum. To serve this part of the customer base, this flexibility can be implemented on the condition that it is hidden from the basic users.

4 Conclusion

A usability experiment was performed to gain insights into the hypothesized negative impact that the flexibility inherent to the implementation of variability has on the usability provided to end-users. Three user tasks were defined and for each task, two variability points with a demonstrable difference in the degree of flexibility provided were selected. Results of the experiment indicate that usability, in both effectiveness and efficiency, is indeed negatively impacted by flexibility. Software vendors that want to extend their potential customer base by increasing the flexibility of their software should be aware of the adverse consequences for usability as a serious decrease in usability causes customers to cancel their subscription. Considering the presumed non-linearity of the curve of the relationship between flexibility and usability, a flexibility- and usability focused optimum can be found using the usability evaluation approach of this study. Once these optima are found, a detailed variability implementation strategy can be defined.

5 Discussion and Future Research

In order to test the extent to which the level of variability affects the usability provided by variability points, we have operationalized usability – the dependent variable of the usability experiment. However, this is contrasting to our approach towards the independent variable (i.e. flexibility). Although we have compared the usability of variability points having a demonstrably difference in the degree of flexibility provided to the end-user, further research is required to develop a more formal approach to describe and simulate this increase in flexibility. A first step towards this purpose is to define the concept of flexibility. Despite the fact that flexibility is hard to operationalize, it is possible to compare variability points offering similar functionality on the flexibility they provide, as was done with the EOL and QB VPs in this study. Future research is required to improve and formalize this relative classification. The usability experiment was set up such that we had three opportunities to compare two variability points with a demonstrable difference in the flexibility they provided to end-users. Despite the fact that from all three cases (i.e. tasks) could be observed that the usability

was lower for variability points providing higher flexibility, the difference between high and low flexibility VPs differed substantially among the different tasks. For the second task for example, we can hardly argue that the difference between the two VPs is significant, even though we have sufficient qualitative data to exemplify this small difference shown by the usability metrics. In this case, the GUI was an important factor affecting the results on the usability measures. This is one of the disadvantages of conducting such an experiment using functionalities of existing software products. One of the advantages though, is that it is simpler to simulate a real life usage environment when using existing (successful) software products. By conducting the usability evaluation sessions according to the question-suggestion protocol, validity threats inherent to the VPs (e.g. when the GUI affects the usability measures) can be counteracted as this protocol facilitates the collection of qualitative data which can provide valuable context to the quantitative metrics. Another factor influencing the results of the experiment is the selection of participants. What happens when participants are under- or overqualified? One can expect that in both directions, differences between VPs become smaller as participants that are overqualified might conduct the task flawless in both the low and high flexibility scenario, and the other way around. The fact that we used an existing software product for the experiment simplified the participant selection, as we could base the criteria on the target group defined by the case company (i.e. the software vendor). Considering the influence that factors like participant selection (and other context of use attributes) and VP characteristics (e.g. the GUI) have on the usability measures, more experiments should be conducted to confirm the outcomes of this study. Besides an observed negative relationship between flexibility and usability, the usability experiment resulted in the presumption that this relationship is amplified by task comprehensiveness. More research is required to gain insights on whether and how task comprehensiveness impacts the relationship between flexibility and usability. Another presumption that needs further research is the shape of the curve of the relationship between flexibility and usability. Based on the usability evaluation sessions, we presume that the decrease in usability due to an increase of flexibility is not linear. Instead, we expect this effect to be amplified as the flexibility increases.

References

1. C.P. Bezemer, A. Zaidman, B. Platzbeecker, T. Hurkmans, and A. 't Hart. Enabling multi-tenancy: An industrial experience report. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1 –8, September 2010.
2. F. Chong and G. Carraro. Architecture strategies for catching the long tail. *MSDN Library, Microsoft Corporation*, page 9–10, 2006.
3. Tovi Grossman, George Fitzmaurice, and Ramtin Attar. A survey of software learnability: metrics, methodologies and guidelines. In *Proceedings of the 27th international conference on Human factors in computing systems*, page 649–658, 2009.

4. S. Jansen, G. J. Houben, and S. Brinkkemper. Customization realization in multi-tenant web applications: case studies from the library sector. *Web Engineering*, page 445–459, 2010.
5. J. Kabbedijk and S. Jansen. Variability in multi-tenant environments: architectural design patterns from industry. *Advances in Conceptual Modeling. Recent Developments and New Directions*, page 151–160, 2011.
6. J. Kabbedijk and S. Jansen. The role of variability patterns in multi-tenant business software. In *Proceedings of the WICSA/ECSA 2012 Companion Volume*, page 143–146, 2012.
7. Christie D. Michelsen, Wayne D. Dominick, and Joseph E. Urban. A methodology for the objective evaluation of the user/system interfaces of the MADAM system using software engineering principles. In *Proceedings of the 18th annual Southeast regional conference*, page 103–109, 1980.
8. R. Mietzner, A. Metzger, F. Leymann, and K. Pohl. Variability modeling to support customization and deployment of multi-tenant-aware software as a service applications. In *Proceedings of the 2009 ICSE Workshop on Principles of Engineering Service Oriented Systems*, page 18–25, 2009.
9. Jakob Nielsen and JoAnn T. Hackos. *Usability engineering*, volume 125184069. Academic press Boston, 1993.
10. A. Seffah, M. Donyaee, R. B. Kline, and H. K. Padda. Usability measurement and metrics: A consolidated model. *Software Quality Journal*, 14(2):159–178, 2006.
11. M. J. Van den Haak, M. D. T. de Jong, and P. J. Schellens. Employing think-aloud protocols and constructive interaction to test the usability of online library catalogues: a methodological comparison. *Interacting with computers*, 16(6):1153–1170, 2004.
12. J. van Gurp, J. Bosch, and M. Svahnberg. On the notion of variability in software product lines. In *Working IEEE/IFIP Conference on Software Architecture, 2001. Proceedings*, pages 45 –54, 2001.
13. C. D. Weissman and S. Bobrowski. The design of the force. com multitenant internet application development platform. In *Proceedings of the 35th SIGMOD international conference on Management of data*, page 889–896, 2009.