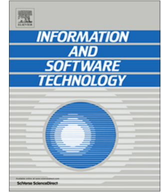
Information and Software Technology 60 (2015) 16–31

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof



An industrial case study on variability handling in large enterprise software systems

Matthias Galster [a,*], Paris Avgeriou [b]

[a] University of Canterbury, Christchurch, New Zealand
[b] University of Groningen, Groningen, The Netherlands

A R T I C L E   I N F O

Article history:
Received 21 September 2013
Received in revised form 27 October 2014
Accepted 10 December 2014
Available online 22 December 2014

Keywords:
Variability
Enterprise software systems
Case study
Grounded theory

A B S T R A C T

Context: Enterprise software systems (e.g., enterprise resource planning software) are often deployed in different contexts (e.g., different organizations or different business units or branches of one organization). However, even though organizations, business units or branches have the same or similar business goals, they may differ in how they achieve these goals. Thus, many enterprise software systems are subject to variability and adapted depending on the context in which they are used.
Objective: Our goal is to provide a snapshot of variability in large scale enterprise software systems. We aim at understanding the types of variability that occur in large industrial enterprise software systems. Furthermore, we aim at identifying how variability is handled in such systems.
Method: We performed an exploratory case study in two large software organizations, involving two large enterprise software systems. Data were collected through interviews and document analysis. Data were analyzed following a grounded theory approach.
Results: We identified seven types of variability (e.g., functionality, infrastructure) and eight mechanisms to handle variability (e.g., add-ons, code switches).
Conclusions: We provide generic types for classifying variability in enterprise software systems, and reusable mechanisms for handling such variability. Some variability types and handling mechanisms for enterprise software systems found in the real world extend existing concepts and theories. Others confirm findings from previous research literature on variability in software in general and are therefore not specific to enterprise software systems. Our findings also offer a theoretical foundation for describing variability handling in practice. Future work needs to provide more evaluations of the theoretical foundations, and refine variability handling mechanisms into more detailed practices.

© 2014 Elsevier B.V. All rights reserved.

1. Introduction

1.1. Research problem

Variability is the ability of a software system or artifact (such as a component or subsystem) to be configured, customized, or extended for use in a specific context [1]. Thus, variability supports multiple deployment- or operation scenarios of a software system. To enable variability, parts of the system are not fully defined during early iterations of development, but later when more details about a specific customer or concrete usage scenarios are known. However, during early iterations, software engineers need to identify what parts of a system should be variable (e.g., in terms of "variation points" in the design) and decide how this variability can be resolved (e.g., in terms of "variants"). Later, these variants are chosen to resolve variability at a variation point.

In this paper, "handling variability" includes (a) describing variability (i.e., expressing potential variants) and (b) resolving variability (i.e., implementing or selecting variants at a variation point). This notion of "handling variability" includes that of "realizing variability" as used by Svahnberg et al. [2] who define "realizing variability" as implementing a variation point. It is also similar to the notion of "managing variability" as used by van der Linden et al. [3], which includes defining and implementing variability, among other things. Systematically handling variability leads to less design and development effort due to higher reusability of software artifacts, as well as reduced maintenance costs compared to designing separate systems for diverse usage scenarios [3].

Many of today's software systems are built with variability in mind, e.g., product lines or families, self-adaptive, configurable or customizable single systems, open platforms, or service-based

* Corresponding author.
E-mail addresses: mgalster@ieee.org (M. Galster), paris@cs.rug.nl (P. Avgeriou).

http://dx.doi.org/10.1016/j.infsof.2014.12.003
0950-5849/© 2014 Elsevier B.V. All rights reserved.

systems that support the dynamic composition of services [4,5]. One prominent category of variability-intensive software systems are enterprise software systems (ESS). In our work, we adopt Fowler's notion of ESS as applications that "are about the display, manipulation, and storage of large amounts of often complex data and the support or automation of business processes with that data" [6]. Typical functions of ESS include order processing, procurement, production scheduling, customer information management, or accounting. ESS are hosted on servers and provide simultaneous services to a large number of users over a computer network (e.g., in contrast to single-user applications that run on personal computers and serve one user at a time). ESS are often designed to run in different business units or branches of an organization (each with their specific requirements), or in different countries in which a company operates (each with specific constraints on business processes). For example, branches of an oil and gas business in different countries serve the same purpose and have to provide the same or similar services, but are subject to different regulations, laws, staff structure, etc. Also, enterprise software systems are used by many different customer types (e.g., oil and gas, retail). These differences affect business processes and functionality of enterprise software systems, and eventually the implementation of functionality. Therefore, ESS are large-scale systems and support a broad range of features. Implementing one system that accommodates all possible variants of these features would increase the size and complexity of already very complex systems exponentially with the number of features and variants. Thus, handling variability in a system (either single systems, product lines, system of systems, etc.) early on is preferred over discovering and addressing variability later in the product life cycle [7]. Moreover, as variability is pervasive and affects many stakeholders and development activities, software engineers need proper support in the form of methods and tools for handling variability [8].

1.2. Research questions and contributions

This paper aims at understanding how variability is handled in large enterprise software systems in practice. We study enterprise software systems as they have a profound impact on an organization, and variability needs to be handled in both the software and the organization. Furthermore, ESS are built for a broad range of customers and used in many different types of organizations (e.g., large or medium-sized companies, different domains), and therefore must run in many different environments, usage contexts, etc. Ignoring variability during design and designing individual systems for each environment, usage context, etc. can become prohibitively expensive at later stages of the product life cycle when it becomes more difficult to adapt systems. In detail, defining the study goal in terms of GQM [9], the goal of the case study presented in this paper is to

– *analyze* types of variability and mechanisms,
– *with respect to* handling variability,
– *for the purpose of* improving software development theory and practice,
– *in the context of* variability-intensive enterprise software systems,
– *from the viewpoint* of practitioners.

The *types* of variability characterize variability in terms of the artifacts in which variability occurs. For example, variability can occur in conceptual artifacts (e.g., specification of functionality), or parts of the ESS implementation (e.g., backend systems, data, external interfaces, internal interfaces or operating systems), or artifacts at the business level (e.g., business processes supported by an ESS). By *mechanisms* to handle variability we mean mechanisms to describe and resolve variability (see our definition of handling variability in Section 1.1).

To operationalize this goal, we define two research questions:

– **RQ1: What types of variability occur in a large-scale enterprise software system?** Answering RQ1 is the first step of analyzing variability by providing a list of variability types that need to be handled in large-scale ESS. Each type represents a different kind of artifact where variability can be manifested. In our study, types of variability emerged as part of the grounded theory data analysis. Different types of variability may be addressed by different mechanisms, which leads to RQ2.
– **RQ2: What mechanisms exist to handle variability in large-scale enterprise software systems?** Mechanisms that software organizations use to handle variability can be considered as good practices that might be reusable when designing new systems. Therefore collecting and describing such mechanisms can support practitioners in reflecting about and eventually selecting the optimal ways to handle variability in their context. Furthermore, since variability handling mechanisms also emerged following a grounded theory approach, answering this research question helps us compare existing theories for handling variability (e.g., Svahnberg et al. [2]) with actual industrial practice (see Section 6.1.3 for a comparison).

To achieve the aforementioned goal, we present a case study in two large software organizations that develop large enterprise software systems. As a result of the case study, the paper contributes types of variability in enterprise software systems and mechanisms to handle variability, as well as an in-depth analysis of those types and mechanisms. Our results describe a picture of industrial reality and provide empirical evidence for how variability in ESS is handled in practice [10]. In contrast to other research that discusses types of variability (such as Chang and Kim [11], Segura et al. [12]) or variability management (such as Svahnberg et al. [2], Sinnema and Deelstra [13]), we do not propose or advocate approaches, but capture current practice. So far no empirical evidence for handling variability in ESS exists (see also Section 2).

The target audience of this paper is twofold: First, we aim at practitioners who would like to find out about good practices that they can add to their toolbox for handling variability in enterprise systems. Furthermore, practitioners can benchmark their own variability practices. Second, our results provide researchers with a view on how variability is handled "in the wild". This helps identify how research themes and approaches in practice complement each other, or if there are significant gaps between research and practice that need to be filled. Finally, our results provide researchers with a theoretical contribution in terms of a classification of variability types and variability handling mechanisms based on empirical evidence, including a reflection to existing concepts.

1.3. Paper structure

In Section 2 we briefly discuss the background of our research and summarize related work. Section 3 presents the research methodology. Section 4 presents the results of our study, which are further analyzed in Section 5. In Section 6 we discuss our results and strengthen the theoretical foundation by comparing our findings with similar literature. In Section 7 we acknowledge limitations of our work before we conclude in Section 8.

2. Background and related work

This section briefly presents some background information on variability and related work on variability in practice. When using

a grounded theory approach for data analysis, extensive literature reviews in the same area of research should be avoided to reduce bias during analysis (grounded theory analyses data without having findings in mind) [14]. Thus, this section only presents a minor review of related concepts.

2.1. Variability

Variability as the ability of a software system or artifact to be changed or adapted in a planned manner is primarily studied in the software product line domain [15]. For example, variability in enterprise software systems from a product line perspective has been investigated in [16,17]. Product lines describe variability explicitly in terms of features and decisions, and encompass conceptual models, such as feature models, decision models or component-and-connector models [18]. A product line assumes the existence of a product line infrastructure, including related processes (e.g., core asset development, product development, management) [19]. As found in our previous work, this is rarely the case for many ESS which are subject to variability [20,21].

2.2. Variability in practice

A study conducted by Chen and Babar had a similar goal to our study (study variability in practice), but the former investigated *challenges* related to variability management in industry (e.g., documentation or testing) [22], whereas we identify industrial *practices*. Similarly, Ihme et al. studied what current variability challenges exist in small and medium enterprises and what the practices are in small and medium sized enterprises to manage variability [23]. The authors conducted a multiple case study using semi-structured interviews and identified technical challenges (e.g., decision making, change implementation) and non-technical challenges (e.g., resourcing, business models). Cases were software companies, whereas in our study, cases are software systems. Furthermore, we focus on large-scale enterprise systems whereas Ihme et al. studied variability management in small and medium organizations.

Jaring and Bosch analyzed variability in an industrial software product line to identify variability-related issues (variability identification, variability dependencies and tools support) and to propose a representation of variability realization mechanisms based on introduction and binding of variability (e.g., at runtime, installation) [24]. Our study is complementary to this work in that we (a) investigate large-scale ESS, and (b) do not focus on product lines.

An industrial survey with 25 software engineering practitioners described the current state of practice of variability management in 25 small and medium-sized enterprises (SME) in Sweden [25]. This survey concludes that SME can benefit from systematic variability management but that there is a lack of awareness for systematic variability management. In particular, little reuse and traceability activities are performed. Again, our study is complementary to this survey as we investigate large-scale enterprise systems and do not focus on product lines.

3. Research method

Our research is motivated by a practical problem that cannot be studied in isolation from its context. Thus, we apply case study research as an "in-the-wild" method, rather than "in the lab" methods (e.g., controlled experiments). Furthermore, when investigating variability in large enterprise software systems in an industrial context, we have little control over all involved variables (e.g., people, organizational structures, politics). Finally, case studies are a good way to get in-depth understanding of current industrial practices and solutions by allowing "how" and "why" questions [26].

We use a multiple-case, embedded case study [27]. "Multiple case" means that we study two cases (i.e., two large-scale enterprise software systems). "Embedded" means that we have multiple units of analysis that are observed in the two cases (types of variability, mechanisms to handle variability). Case studies might be descriptive, explanatory, exploratory or evaluatory [28]. We follow the exploratory approach [29] because exploratory studies are used in cases where "research looks for patterns, ideas, or hypotheses rather than research that tries to test or confirm hypotheses" [30]. We currently cannot form hypotheses because we still lack an understanding of how variability in enterprise software systems is handled in practice; thus the study is not explanatory. The study is also not descriptive since our study does not start with an existing theory or model. The case study design follows the guidelines proposed in [31].

3.1. Study design and execution

3.1.1. Case selection

To select the two cases for the case study, we applied a "crucial case selection". Crucial cases are considered representative cases in a certain domain [32]. Criteria for selecting our two cases were as follows:

- To achieve our research goal, we aimed at *large-scale and complex* enterprise software systems (e.g., many components, integration with existing infrastructures and IT systems), rather than consumer software (e.g., desktop applications with the same functionality for all users, such as word processing software) or embedded software.
- We looked at *variability-intensive* enterprise software systems, i.e., systems that required a high degree of adaptability. In both selected cases, variability between customer organizations is too big to provide one system with the same functionality for all customers. On the other hand, developing and maintaining individual solutions for each customer requires significant effort. The developed software in both cases is deployed and used in several system variations and used by a broad range of customers (System 1) or in different business units of the same organization (System 2).
- The software should not be part of a dedicated product line. As we argued in Section 2, many ESS which should support variability are not supported by a dedicated product line infrastructure (i.e., product line processes, dedicated application engineering and domain engineering, core asset bases, etc.); both selected cases also fulfill this criterion.

Differences in the two cases lie in the application domains, the number of customers that currently use the products developed in the studied cases, the detailed software development processes followed, and the type of end users. For confidentiality reasons we cannot report details about the systems under study or the organizations that develop the systems, but we provide a brief overview of the cases and organizations in the following subsection. Furthermore, for confidentiality reasons, some terminology used within Organization 1 for certain concepts or techniques has been changed.

3.1.2. Organization 1 and case System 1

System 1 is a large-scale enterprise resource planning (ERP) system developed by a large software organization (Organization 1). System 1 has been developed for more than 15 years, in several releases. Each release consists of several million lines of code.

Due to its size, customers of Organization 1 rarely completely replace System 1 with a new release, but rather update their system. System 1 is offered as a core ERP system with several industry-specific solutions (e.g., for oil and gas, retail). Several modules support key functionalities, such as supplier/customer relationship management, product lifecycle management, supply chain management or financial services.

Variability in System 1: System 1 is variability-intensive because it is used by a broad range of customers. Variability in System 1 appears in two dimensions. First, a variant of System 1 is offered for different domains (e.g., oil and gas, education). Second, within each domain, each customer that uses System 1 is different and, given the size and organizational impact of System 1, has its own detailed requirements for how to implement business processes. When delivering new features for System 1, the core business logic originally implemented for a customer should remain stable. New features should be delivered to the customer without interrupting its business. In the early days of System 1, it was not possible to get a new version of System 1 without also implementing new versions of business processes. More recent versions of System 1 acknowledge that customers are different, with different requirements, and allow System 1 to be adapted to existing business processes.

Software development process for System 1: In Organization 1, a company-wide product life cycle describes how software is developed. The product life cycle is agile rather than strictly sequential. It defines several milestones and sprints. Guidelines for the whole architecture of System 1 are not feasible as individual modules of System 1 are too diverse. Thus, guidelines exist to support decision making during systems design by providing many decision alternatives that can occur in individual products of System 1. These guidelines define the range of products and rules for all products derived for System 1. However, the guidelines do not include all possible decisions and do not include open decision topics but rather high-level scoping (e.g., new product version needs to work with a certain version of a certain product; for a certain type of product certain decisions should be made, some with predefined options). The guidelines are defined by one or more chief architects and used by developers who instantiate a product.

3.1.3. Organization 2 and case System 2

System 2 was a project for a large global customer of Organization 2. We requested approval to disclose details about the project, but the customer did not agree to release its name or any project details that could reveal its identity. The project we studied was a web selling platform to sell a broad variety of high value products in a business-to-business (B2B) context. The products sold on this platform are very diverse. The platform helps the customer achieve a market share of around 40% and generates several billions of Euro revenue per year. At the time we studied System 2, it was undergoing its second major evolutionary redesign in 11 years, i.e., the system has been in use and evolving for more than 11 years. The first redesign took place in 2003 with variability being one of the main reasons for the redesign. One major decision was to move toward a SOA-based solution. The redesign involves 10–12 developers and four architects as technical lead. The redesign has been scheduled to last 9 months.

Variability in System 2: System 2 is variability-intensive as it needs to support many variation points for many business units (e.g., backend systems). Also, there are different roll-outs of the system (e.g., for each region in which the customer is active). Causes for variability in the B2B system are mainly the requirements for the customer and the processes affected by these requirements. Furthermore, different IT landscapes exist in business units of the customer. We show variation points and variants that occur in System 2 in a high-level domain model of System 1 in Fig. 1 (see Section 4.1).

Software development process for System 2: The development process followed in Organization 2 to develop System 2 only includes fundamental activities, but it can be used as a full process to build a system, or activities can be added or tailored as needed. Organization 2 follows an incremental and iterative process, and utilizes use cases and scenarios to define requirements. Close collaboration amongst different stakeholders (including continuous feedback) helps align different interests. In contrast to System 1 and Organization 1, there are no central design guidelines. Instead, guidelines depend on constraints and development requirements imposed by customers. The design process for System 2 is a combination of bottom-up (domain composition, e.g., identifying functional domain entities, such as shopping catalog) and top-down (business process decomposition). A product line engineering approach was not considered as the system is developed for one customer with different deployment contexts.

3.1.4. Data collection techniques

We include different data sources in order to obtain more precise data. We collected qualitative data using the following data collection techniques and data sources [31]:

- Direct technique: We conducted seven semi-structured interviews. The interview guidelines are included in the appendix. Participants of interviews had 10–20 years of industrial software engineering experience. We obtained only partial consent to record interviews, i.e., only some interviews could be recorded. For all interviews, we took extensive notes and copies from in-depth whiteboard discussions. For example, for System 2 we modeled the business process underlying the B2B platform, including its variability concerns that need to be addressed and stakeholders that have an interest in these concerns. We used a semi-structured interview with open questions to allow a broad range of answers and issues to be discussed. Questions were planned but not necessarily always followed in the same order. The semi-structured interviews allowed improvisation and exploration of topics that arose during interviews. Interviews were conducted at four different locations of Organization 1 and Organization 2 in Germany and Switzerland. Before each interview, introductory letters were sent to participants to discuss the aim of the interview. Interviews lasted between one and three hours. To get feedback from interviewees and avoid misunderstanding, we summarized major findings at the end of each interview. Details about interviewees are shown in Table 1 (minutes in Table 1 show the minutes recorded, not the duration of the interview).
- Indirect techniques: We studied technical reports, process descriptions and development resources at Organization 1 and Organization 2. Most documents were internal documentation not for public release. For example, we studied architecture documentations, internal developer community networks and software development guidelines in Organization 1, and business process descriptions from customers in Organization 2.

3.2. Data analysis

As we conducted an exploratory case study that resulted in qualitative data, we used a grounded theory approach for data analysis [33]. Grounded theory analyses data without having findings in mind. Also, grounded theory does not search for confirmations of previous findings [34]. The answers to our research questions emerged by constantly comparing indicators found in the data to previously identified indicators. Indicators are actual data, such as behavioral actions and events observed or described

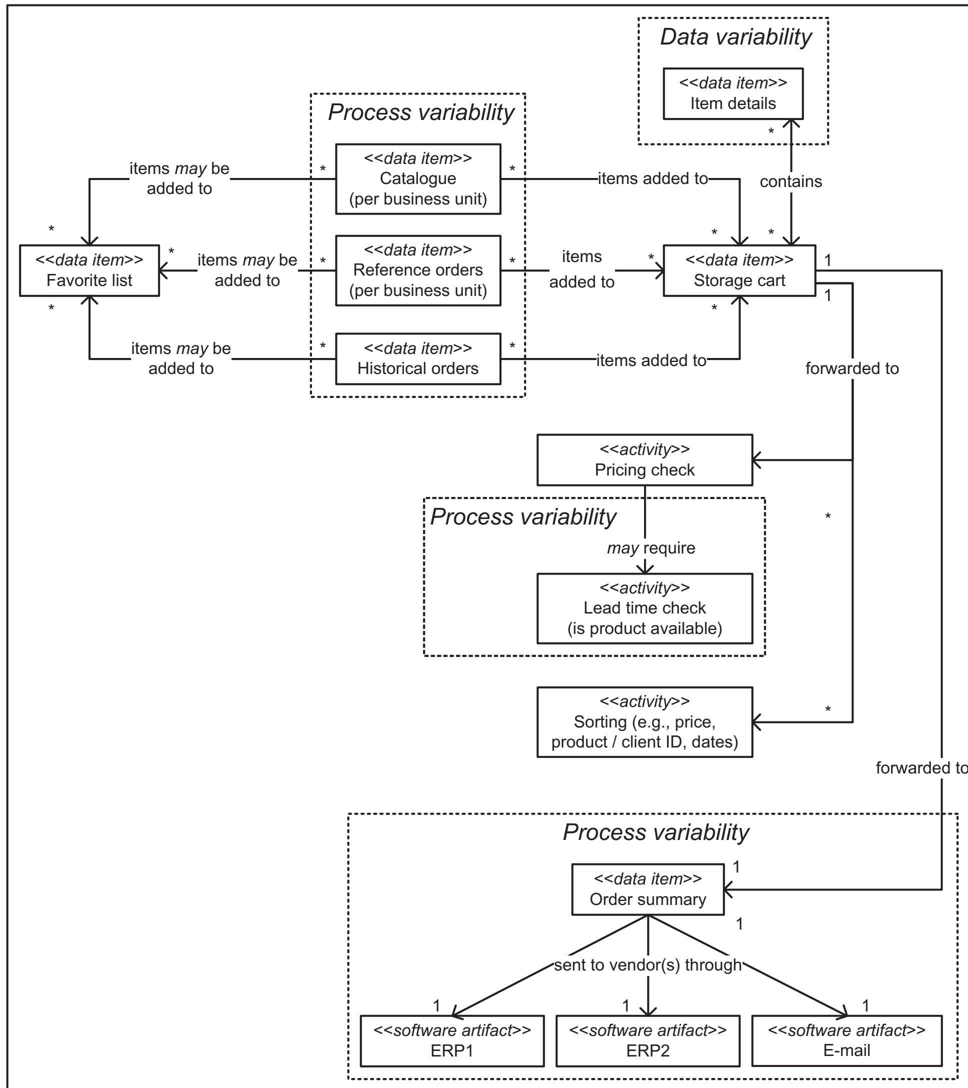


Fig. 1. Domain underlying System 2.

Table 1
List of interviewees.

Role	Recorded (yes/no)	Notes (# of pages)	Organization
Product researcher	Yes (~60 min)	11	1
Product researcher	Yes (~60 min)	11	1
Lead product researcher	Yes (~60 min)	11	1
Product architect	Yes (~65 min)	13	1
Architect	No	13	1
Master data manager	Yes (~50 min)	7	1
Senior IT architect	No	16	2

in documents (documentation, online resources of companies involved in the case study) and in interviews. An indicator may be a word, a phrase, a sentence, or a paragraph in the data [35]. A finding is either supported by additional indicators in the data, or rejected if no indicators are found in the data. In detail, the following four data analysis steps were performed:

1. **Pre-processing**: We transcribed recorded interviews (for interviews that were recorded) and rewrote notes made during the interviews (for all interviews). Furthermore, we added manual notes to all transcripts to obtain a full record for each interview.
2. **Coding**: Codes are used to name indicators [35]. We used a simple method of open coding [36] where one code can be assigned to many pieces of text, and one piece of text can be assigned to more than one code [37]. During the analysis, this set of codes evolved. All interview transcripts and documents were thoroughly read, and phrases of interest were coded to reflect the topic of that phrase [38]. In a first phase of coding, we looked at one transcript from one participant to identify different codes. These codes emerged from the data in the transcript as well as from the questions we had on our questionnaire. In a second phase, these initial codes were used as a reference to

code all transcripts. We tried to assign every statement in the transcripts to a code. No new codes were discovered during that phase.

3. **Identification of concepts**: Concepts are abstractions suggested by indicators [35]. After initial coding, we looked at groups of code phrases and merged them into concepts. As our data was collected within a case study, the data is context sensitive. Thus, we performed iterative content analysis to make inferences from collected data in its context [39]. Constant comparison was used to generate concepts from indicators and to analyze our data to generate categories of data [38]. Analyzing qualitative data required integrating data where different participants might have used terms and concepts with different meanings, or different terms and concepts to express the same thing [40]. To address this problem, we used reciprocal translation [40]. Reciprocal translation helps summarize newly identified concepts that relate to other similar concepts by translating similar concepts into one another.
4. **Categorization of concepts**: Concepts from different sources were compared to identify categories of concepts, as findings that occurred in more than one data source are more reliable. Also, we investigated if some concepts were sub-issues of other concepts. For that reason, line of argument synthesis was used to identify the "main theme" of different concepts [40]. Line of argument synthesis could be applied to all concepts in an iterative manner until higher level concepts were not overlapping and no sub-concept was part of another higher-level concept. After several iterations, concepts described types of variability and means to handle variability.

4. Results

4.1. RQ1: What types of variability occur in a large-scale enterprise software system?

For this research question we analyzed data coded and categorized as "variability type" that describe levels and artifacts of variability. For System 1, we found seven types of variability. For System 2 we found four types of variability, which overlap with those of System 1. Thus, we identified seven distinct types of variability. The following subsection elaborates on these seven different types of variability. Table 2 maps the variability types to System 1 and 2.

4.1.1. Types of variability

VT1 – Variability in functionality: Variability in functionality is mainly caused by different customers (System 1) or business units (System 2). For example, in System 1, customers in the oil and gas industry have different procurement requirements than customers in the healthcare domain and therefore require different implementations of procurement functionality (e.g., different taxation functionality, different types of products that are delivered as goods or services). Variability in functionality is usually resolved at one point in time (i.e., a variant is chosen once).

Table 2
Variability types.

Variability type	System 1	System 2
VT1 – Variability in functionality	✓	✓
VT2 – Variability in backend systems	✓	✓
VT3 – Variability in data	✓	✓
VT4 – Variability in operating systems	✓	
VT5 – Variability in external interfaces	✓	
VT6 – Variability in user interface	✓	
VT7 – Variability in process	✓	✓

VT2 – Variability in backend systems: Backend systems for ESS are (often mandatory) software systems that support the ESS. For example, backend systems could be different database systems, or may retrieve data from other ESS that communicate with System 1 or System 2. A backend system for System 1 is a database system that stores operational data. Major databases are supported by System 1 but customers can choose their database management system. Backend systems for System 2 include different ERP systems that interact with the B2B platform. Each business unit where System 2 is deployed has different backend systems.

VT3 – Variability in data: Besides variability in backend systems, there is also variability in the actually required and processed data and data structures in ESS. Each customer might have different data tables and dependencies between tables defined in the database. For example, for document management in System 1, customers can decide what document types should be handled and how workflows are organized based on the document type. Further examples for data variability in System 1 are different data required for customer relationship management, or different data required to describe the production process of a customer. In case of System 2, data variability mainly refers to data variability in the product catalog used in the B2B platform. The data structure for the product catalog is based on the business unit where a catalog is used and created.

VT4 – Variability in operating systems: Variability in operating systems refers to the different platforms on which the ESS is supposed to operate. For example, System 1 must work with all major operating systems (different versions of Microsoft Windows and Unix/Linux-based operating systems). Customers can choose the operating system. In contrast to VT2 (variability in backend systems), variability in operating systems refers to platforms on which an ESS will run, rather than other (mandatory) backend systems the ESS communicates with.

VT5 – Variability in external interfaces: "External interface" refers to interfaces required to communicate with any kind of entity (software, hardware, etc.) that is outside the scope of the organization that runs the ESS (e.g., an oil and gas provider communicating with a corporate customer). For instance, an interface to a backend system (e.g., to another ESS) would be "external" if the other ESS is outside the organization of the ESS. Given different external entities that interact with an organization, the connections to external organizations (e.g., connection between companies or consumers connecting to companies) vary. For example, System 1 interacts with many external entities to execute business processes in organizations.

VT6 – Variability in user interface: Different customers might want to have different "user-visible" interfaces. Variability in user interface is often not only due to differences in functionality but also caused by resource constraints. Also, variability in user interface reflects preferences of different users in how actions should be performed. For example, an ESS can provide the same functionality but the user interface layout differs depending on whether the application is accessed through a mobile device or a desktop application. In case of System 1, a procurement module would provide more sophisticated search features to users of the desktop application compared to users of mobile devices (mobile devices may require a more light-weight user interface due to resource constraints of the device). Since parts of System 1 are cloud-based, the functionality of the procurement module is the same for all users, but not all users have access to these functionalities due to different user interfaces. For example, in a purchasing process, the availability of a product may be checked *before* or *after* entering the required amount of items to be ordered; thus, the ordering form would look different for checking availability *before* or *after* even though both provide the same functionality.

VT7 – Variability in process: Processes are usually the means to achieve functionality, i.e., processes underpin functionality (see also Fig. 2 and Section 5.1 where we relate variability types to each other). These processes can vary. In contrast to VT1 (variability in functionality), VT7 is more fine-grained since functionality is achieved through processes. For example, in System 1, business processes for the ERP system differ (e.g., organizations in the banking industry have specific HR procedures and legal constraints, compared to organizations in the automotive industry). Looking at System 2 we found that the original goal of System 2 was to unify all processes in business units. However, processes were too diverse as data access across business units and processes differed (e.g., processes to access external ERP systems in business units is specific to business units). Another example for variability in process in System 2 is due to different order types: Historical orders are orders submitted by phone or fax and then entered into the B2B system; reference orders on the other hand are orders created directly in the B2B system by a customer.

4.1.2. Summary

Table 2 summarizes the variability types found in enterprise systems and relates them to the case in which we found them. We note that these variability types are not specific to enterprise software systems, but also occur in other application domains. For example, variability in functionality typically occurs in product lines [15], or variability in processes occurs in service-based systems [20]. In summary, considering variability-related characteristics of our subject of study (ESS) and focusing the scope and applicability of our constructs for RQ1 (types of variability) we conclude that ESS have to support the diverse needs of different customers and domain-specific functions (VT1), have to handle large amount of complex and different types of data (VT3), have to integrate with other applications and entities within the organization (VT2) and outside the organization (VT5), have to support different business processes (VT7) and run on different platforms (VT4). Furthermore, ESS may require different user interfaces (VT6).

To provide a more concrete example of some variability types, Fig. 1 illustrates parts of the domain underlying System 2 as extracted as part of the case study. Since Fig. 1 is a domain model, it does not fully describe business processes but shows the major domain elements and their relationships, including multiplicities. Dotted boxes in Fig. 1 indicate the scope of a variability type. Solid boxes show domain elements with stereotypes (e.g., data items, software artifacts). Arrows show relationships between domain elements. The dotted boxes in Fig. 1 show two of the four types of variability for System 2 (VT3 – Variability in data, VT7 – Variability in process). Process variability (VT7) occurs because orders are either placed based on a catalog (which differs between business units), or because they are historical orders (through the phone). Also, process variability occurs in the way that orders are submitted to product vendors (e.g., orders could be submitted through e-mail or by directly connecting to ERP systems of vendors). "may require" between pricing check and lead time check indicates that lead time check is not always required (i.e., this would be another instance of process variability). Data variability (VT3) occurs because the description of items (e.g., products) varies.

4.2. RQ2: What mechanisms exist to handle variability in large-scale enterprise software systems?

To answer this research question, we synthesized data labeled with "variability handling". The following subsection elaborates on the different mechanisms to handle variability while Table 3 maps the mechanisms to System 1 and System 2. Note that there may be more approaches for handling variability in software systems in general. However, we did not find any indication of these being used in the context of our case study.

4.2.1. Mechanisms to handle variability

In this section, we list all mechanisms that we found are used to handle variability. Some of the mechanisms are not specifically aiming at handling variability per se, but support software development and maintenance in general. However, the goal of our study was to find mechanisms to handle variability. Thus, we highlight how these mechanisms support handling variability. Furthermore, for each variability handling mechanism we explain at the very end, how it is related to handling variability, i.e., describing and resolving variability.

VM1 – Enhancement package concept: Changes to the business of a customer of an ESS usually cause problems to the ESS system of that customer (e.g., for System 1, changes in employees and their responsibilities require new processes to be supported by an ERP without affecting mature processes too much). Furthermore, customers do not want to install every new release of the ESS system. On the other hand, maintaining old versions can be very expensive. The enhancement package concept allows a customer to download new software packages for the ESS. The customer can then decide what new business functions to activate. The "enhanced" software then behaves like the old software but includes new processes (e.g., for System 1, a new "order-to-cash"

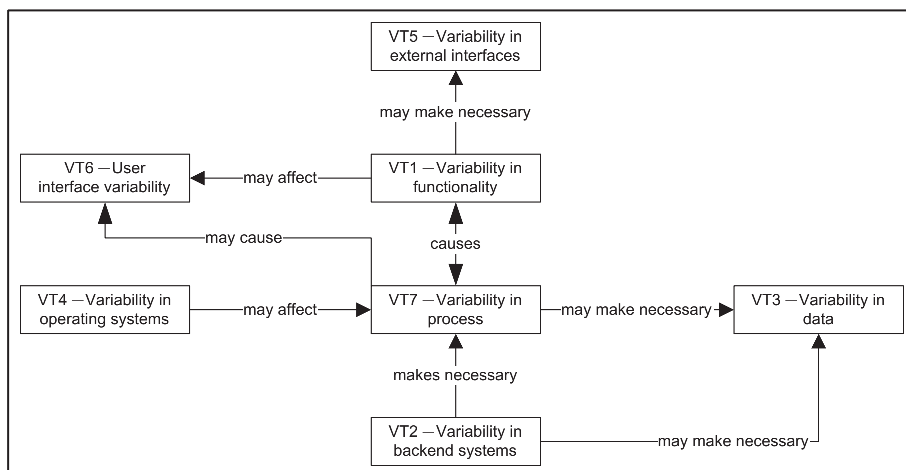


Fig. 2. Relationships between variability types in ESS.

Table 3
Mechanisms to handle variability.

Mechanisms to handle variability	System 1	System 2
VM1 – Enhancement package concept	✓	
VM2 – Add-ons	✓	
VM3 – Function switching	✓	
VM4 – Layering	✓	
VM5 – Add-ins	✓	
VM6 – Metadata modeling	✓	✓
VM7 – Business rules		✓
VM8 – Code switches		✓

process or new process variants could be activated). The technical details of adding these new processes are hidden from the user who makes changes to the business process. For example, in System 1, the enhancement package concept was introduced four to five years ago. The enhancement package concept *describes* variability in terms of different packages and *resolves* variability by activating packages (i.e., business functions).

VM2 – Add-ons: Add-ons allow customized extensions of an existing system. In contrast to add-ins (VM5), add-ons do not require changes in the source code as they are added "on top" of the system (e.g., through high-level API's) rather than "in" the system (i.e., the source code). Thus, add-ons *describe* variability in terms of different extensions and *resolve* variability by selecting extensions.

VM3 – Function switching: All predefined functions and information for all types of systems and industries are delivered to all customers (complete source code). Customers then decide what functions to activate. Function switching does not require access to source code. Instead, configuration tables are selected when instantiating an ESS. This instantiation and activation of specific functions can only be done once. This is because when activating functions, internal dependencies are created. For example, in System 1 function switching defines functions that could be activated for a certain industry (e.g., functions only for organizations in the oil and gas industry). System 1 is delivered to customers as a neutral system, covering industry-specific solutions. A customer then decides if such solutions should be activated. Customers can also decide to use all functions offered by System 1 as a generic system (e.g., HR, ERP). Function switching means that variability is *described* in terms of configuration tables and *resolved* by activating (or selecting) settings in configuration tables.

VM4 – Layering: Layering allows for high-level separation of concerns. For example, in System 1 the lowest layer contains core functionality, such as database access or the implementation of the domain- and system-specific programming language. On higher layers, generic functions that can be used in all instances of System 1 are located. Finally, top layers include industry-specific functionality. Layering also helps limit variability in the database used by a customer: Lower layers handle database related issues. For example, independent programming levels use general SQL statements that are translated into specific database calls for a specific customer (e.g., ODBC, JDBC). This means, layering allows *describing* variability in terms of layers and *resolving* variability by allocating components to layers.

VM5 – Add-ins: Add-ins allow customers to enhance the system by modifying the original source code, without the involvement of the software development organization. Updates would not overwrite modifications of the original source code made by the customer. This is because add-ins provide specific points in the source code where designers anticipated customer-specific modifications (e.g., additional validity checks, or specific database access). These points are foreseen at design time and made explicit in the architecture. From a software development perspective, these points could be considered as empty sub-routines (functions, procedures, methods, etc.). During customization, customers can fill these sub-routines with their own code. For example, add-ins in System 1 and have been used for at least ten years. Add-ins *describe* variability in terms of configuration points in the code and *resolve* variability by manually editing these configuration points with customized source code.

VM6 – Metadata modeling: Metadata modeling allows metamodeling of catalog data in meta-catalogs. For example, in System 2, a meta-catalog describes the data that would be applicable for all products. On the other hand, some data are specific to the type of product and therefore not prescribed in the meta-catalog. Metadata modeling *describes* variability in terms of different data structures and *resolves* variability by selecting data structures.

VM7 – Business rules: Business rules define the most common variants of a process flow (often in terms of "if–then") to influence how a business process is executed. For example, System 2 resolves variability through the configuration of rules for individual customers (e.g., minimum quantity of ordered items per order, or order lead time). Business rules *describe* variability as different flows and *resolve* variability by following a specific flow.

VM8 – Code switches: Code switches are pieces of code that select variants depending on the version of the ESS. In contrast to VM3 (function switching), VM8 does require access and manipulation of source code. Also, in contrast to VM5 (add-ins), code switches do not require the implementation of code in terms of sub-routines but would execute pieces of code depending on the version of the ESS. For example, in System 2, code switches are enabled through the SPRING component plug-in framework for Java, or through a "Displaced Dispatcher" class inside the code that helps select an XML configuration and what service to use, depending on the version of the B2B system. Code switches *describe* variability in terms of different code fragments and *resolve* variability by selecting pieces of code using external invocation mechanisms (e.g., SPRING).

4.2.2. Summary

Table 3 summarizes mechanisms to handle variability and relates them to the system for which we identified the mechanisms. We present a more detailed analysis and classification of these types in Section 5.2. Table 3 shows that there is only one mechanism that is shared between the two systems. Thus, it could be argued that a third case company could let to yet another set of variability handling mechanisms. However, we used grounded theory for data analysis. In the context of a grounded theory-based analysis, Glaser argues that a good theory is characterized by modifiability, i.e., the ability to include variations in emergent categories caused by new data [41]. Thus, Table 3 may be extended with new variability handling mechanisms if data from new cases become available.

4.3. Relating findings to sources and data collection techniques

Table 4 relates variability types and mechanisms to handle variability to the techniques that we used for data collection (Section 3.1.4). This is to show that all findings of our results can be traced to more than one source, hence establishing data source triangulation. "Other" in Table 4 includes developer community networks and business process descriptions. A finding might originate from more than one participant of the interviews, or may appear in more than one type of documentation.

5. Analysis

5.1. Relationships between types of variability

In Fig. 2 we model the relationships between variability types in variability-intensive large ESS. We identified these relationships by

Table 4
Mapping of findings and data sources.

Finding	Interviews	Documentation		
		Architecture	Development guidelines	Other
Variability type				
VT1 – Variability in functionality	✓	✓	✓	✓
VT2 – Variability in backend systems	✓	✓	✓	✓
VT3 – Variability in data	✓	✓	✓	
VT4 – Variability in operating systems	✓	✓	✓	
VT5 – Variability in external interfaces	✓	✓	✓	
VT6 – User interface variability	✓	✓	✓	
VT7 – Variability in process	✓	✓	✓	✓
Variability handling mechanism				
VM1 – Enhancement package concept	✓		✓	✓
VM2 – Add-ons	✓	✓	✓	✓
VM3 – Function switching	✓		✓	✓
VM4 – Layering	✓	✓	✓	✓
VM5 – Add-ins	✓	✓	✓	✓
VM6 – Metadata modeling	✓		✓	✓
VM7 – Business rules	✓	✓	✓	✓
VM8 – Code switches	✓		✓	✓

analyzing what variability types occurred together. Note that many relationships state "may". This is because one variability type may cause another one, or may be affected by another variability type; however, it is not always the case. We defined the following relationships:

– Variability type A "makes necessary"/"may make necessary" variability type B: denotes that variability type B becomes necessary because variability type A exists in a system. For "may make necessary" this means that A often does not exist without B, i.e., B may be a prerequisite for A (or, in case of "makes necessary", A does not exist without B, i.e., B is a prerequisite for A). "makes necessary" also describes cases in which accommodating variability type A requires another variability type. VT1 may make VT5 necessary because variability in functionality may require that different external interfaces are supported to integrate different functionalities from different external providers. VT2 makes VT7 necessary because different backend systems (e.g., data stored in dedicated databases, XML files, or plain text files) may affect how a business process is performed (thus, requiring different versions of a business process). Furthermore, VT2 may make VT3 necessary since different backend systems may require different types of data, data formats, etc. VT7 may make VT3 necessary because different versions of a business process may require the inclusion of different data.
– Variability type A "may affect" variability type B: denotes that the way variability type B occurs depends on variability type A. In contrast to "makes necessary", this is a more loose relation in the sense that the existences of A and B do not depend on each other. VT1 may affect VT6 because user interfaces may appear differently, depending on variability in functionality (for example, a user interface may have different buttons, or different numbers of buttons, depending on what functionality it should support). Similarly, VT4 may affect how VT7 occurs since variability in operating systems may affect the degree to which variability in a process occurs (e.g., if wrapper processes are required to use certain operating system functions): A wrapper process (VT7) may be required to accommodate variability in operating systems (VT4).
– Variability type A "causes"/"may cause" variability type B: denotes that variability type B would not occur without type A. In this sense, this relationship describes causality without defining any other constraints on variability type A or B. Variability in processes is usually a way to achieve variability in functionality, so VT1 causes VT7. Similarly, variability in business processes can be the cause for functionality (thus, the dual "causes" relationship in Fig. 2). On the other hand, variability in processes may require different user interfaces, thus VT7 may cause VT6. In contrast to "makes necessary" this does not imply that B cannot occur without A. Also, in contrast to "makes necessary", "causes" does not focus on achieving variability type A through type B.

We also found an "is alternative for" relationship between variability types. For example, to accommodate different customers, a system could differ in functionality (VT1) or simply make existing functionality not available to all users by constraining the user interface (VT6). This means, different variability types can occur to achieve the same goal (e.g., address different customers). We did not include this relationship in Fig. 2 because it is orthogonal to the three relationship types defined above and may therefore occur among all variability types. Finally, we did not find that one variability type would exclude other variability types.

In previous work, Jaring and Bosch defined variability dependencies in software product lines [42]. However, these dependencies focused on dependencies between variation points, variation points and variants, and dependencies between variants without considering different types of variability. Also, dependencies in [42] are mostly based on "if–then" relationships. Finally, dependencies in [42] are generic without considering the domain of ESS. Thus, relationships between variability types as found in our work are complementary to Jaring and Bosch in that we define relationships on the level of variability types rather than at the level of variation points and variants. Furthermore, we refine generic "if–then" relationships into "makes necessary", "affects" and "causes" relationships.

5.2. Characterization of variability handling mechanisms

In Table 5 we characterize variability handling mechanisms based on two factors:

– *Who*: We map roles from Svahnberg et al. [2] to variability handling mechanisms identified in our study. These roles include *domain engineers* (engineers involved in designing the generic product), *application engineers* (engineers involved in creating concrete products based in the generic product) and *end users*. This mapping shows who would be using a particular variability handling mechanism.

– *When*: We describe when the mechanisms are used by a domain engineer, application engineer, or end user. Based on [2], a mechanism can be used during architecture design, detailed design, implementation, compilation, linking, or runtime.

As can be seen in Table 5, six variability handling mechanisms occur during architecture design (one of them also occurs during compilation). End users are involved in two variability handling mechanisms. Note that variability handling mechanisms may be categorized differently. However, we characterized mechanisms based on their focus found in our study.

Table 5
Characterization of variability handling mechanisms.

Mechanism	Who	When
VM1 (EPC)[1]	End user	Runtime
VM2 (add-ons)	End user	Runtime
VM3 (function switching)	Application engineer	Architecture design, compilation
VM4 (layering)	Domain engineer	Architecture design
VM5 (add-ins)	Application engineer	Architecture design
VM6 (metadata modeling)	Domain engineer	Architecture design
VM7 (business rules)	Domain engineer	Architecture design
VM8 (code switches)	Application engineer	Architecture design

[1] EPC = Enhancement package concept.

5.3. Relating types of variability and mechanisms to handle variability

In Table 6 we show what mechanisms to handle variability can be used to handle what type of variability. This table shows why a certain mechanism would be used and what its purpose is. We defined the mapping between variability types and mechanisms to handle variability by cross-checking in the data what mechanism occurs in the context of which variability type. For example, for System 2, concepts identified during data analysis (see Section 3.2) related to variability types (such as VT3 – Variability in data) were found to be mentioned with concepts related to variability handling mechanisms (e.g., VM6 – Metadata modeling). Also, we checked how mechanisms and variability types are described in system documentation. For example, for System 1, the architecture document states that business rules for base pricing and net pricing (i.e., VM7) are used to implement different pricing procedures (i.e., VT7).

Table 6 shows that variability in processes (VT7) is addressed most. On the other hand, variability in operating systems (VT4), in external interfaces (VT5) and user interfaces (VT6) are addressed least. VT4, VT5 and VT6 may be addressed least by explicit variability handling mechanisms because independent of dedicated variability handling mechanisms, standards to communicate with backend systems (e.g., XML schema for data base communication), standards to support different operating systems (e.g., byte code that can be interpreted on different platforms) or standards to enable communication with external systems (such as EDI or DCAT standards to exchange product catalog data) may be used implicitly to address these types of variability. Note that VT1 (variability in functionality) and VT7 (variability in processes) are handled in similar ways (except for business rules which only apply for variability in processes) since both refer to similar types of variability, but at different levels of abstraction.

As can be seen in Table 6, variability handling mechanisms can be used together to complement each other. Also, as can be seen in Table 6, there are mechanisms that are quite generic and used to support many different variability types. For example, add-ons (VM2) or layering (VM4) help address all seven types of variability. On the other hand, some mechanisms are more specific and help address only few variability types. For example, business rules (VM7) only relate to variability in process (VT7).

6. Discussion of results

In this section we discuss the research results in light of related work, and present the implications for research and practice.

6.1. Relation to existing literature

We used grounded theory as the analysis approach in our case study. Thus, following grounded theory, the data were first collected and analyzed. Once the findings were grounded and developed based on our two cases, literature on variability in ESS was reviewed. In this section we contrast our findings and the theoretical contribution to other literature and previous theories. Enfolding literature and comparing our findings with conflicting and similar literature is an important step to raise the theoretical level of our work and to sharpen the definition of our constructs (i.e., variability types and variability handling mechanisms) [43]. Furthermore, analyzing relevant literature after analysis helps (a) protect our findings from preconceived notions, and (b) relate research findings to literature through integration of ideas [14].

– We compare variability types with variability layers defined in the FORM method proposed by Kang et al. [44] (Section 6.1.1). Here we found that our variability types refine variability in two of the FORM layers: product capabilities and operating environment. The refinement takes place by providing more detailed and specific descriptions about variability in these two layers. However, our variability types do not cover the other two layers: domain technologies and implementation techniques.
– In Section 6.1.2 we investigate whether variability types occur in the problem or solution space as defined by Gurp et al. [45]. We found that our variability types occur in either problem or solution space but not in both at the same time.
– We reflect our variability handling mechanisms to variability realization techniques proposed by Svahnberg et al. [2] (Section 6.1.3). We found that our variability handling mechanisms are broader and also include aspects related to information (data), process and deployment. Thus, the variability handling mechanisms expand the existing taxonomy proposed by Svahnberg et al.

6.1.1. Variability types versus variability layers

In the FORM method (Feature-oriented Reuse Method) Kang et al. characterize variability via different layers [44]:

– Product capabilities (capability layer): services, operations, presentation, use, attributes, etc.
– Operating environment (operating environment layer): hardware platform, operating system, database management system, network, etc.
– Domain technologies (domain technology layer): methods, theories, etc.
– Implementation techniques (implementation technique layer): algorithms, abstract data types, etc.

This separation is also partially reflected in variability types identified in our study. In Table 7 we map our variability types to layers in feature models of FORM. The mapping in Table 7 shows at what layer a variability type would occur. Note that even though VT3 stems from user expectations (requirements) and is therefore rooted in the problem level, it appears in the operating

Table 6
Types of variability and mechanisms to handle variability.

Mechanism	VT1 – Variability in functionality	VT2 – Variability in backend systems	VT3 – Variability in data	VT4 – Variability in operating systems	VT5 – Variability in external interfaces	VT6 – User interface variability	VT7 – Variability in process
VM1 – Enhancement package concept	✓						✓
VM2 – Add-ons	✓	✓	✓	✓	✓	✓	✓
VM3 – Function switching	✓						✓
VM4 – Layering	✓	✓	✓	✓	✓	✓	✓
VM5 – Add-ins	✓	✓	✓	✓	✓	✓	✓
VM6 – Metadata modeling			✓				
VM7 – Business rules							✓
VM8 – Code switches	✓	✓	✓				✓

environment (including database management system as shown above) defined in FORM since FORM does not separate problem or solution domain. Furthermore, it was not possible to map VT6 (variability in user interface) to any level of FORM. As a consequence, domain technology could not be related to any of our variability types. As can be seen, our variability types refine the capability layer and operating environment layer by providing two specific types of variability in the capability layer, and four specific types of variability in the operations environment layer. Also, none of our variability types relate to lower level layers "domain technology" and "implementation technique" in FORM.

6.1.2. Variability types in problem and solution space

As can be seen in the previous section, variability types are not quite uniform in the sense that they would represent variability at the same level or variability of similar kind. Therefore, we identified variability types in the problem space and variability types in the solution space. More specifically, and referring to levels of variability as introduced by Gurp et al. [45], variability types can be related to user expectations, requirements (both are in the problem space), and architecture, design, or code (solution space):

(1) Variability types in the problem space: Variability in the problem is introduced by customers/end users, and resolved by domain engineers and end users often during requirements analysis. VT1, VT3, VT5, and VT7 tend to address variability in the problem space and in user expectations.
(2) Variability types in the solution space: Variability types in the solution tend to be introduced and resolved by architects (as domain engineers and application engineers), or developers, either during requirements analysis or the implementation stage (architecture, design, code). VT2, VT4, and VT6 address variability in the solution.

In Table 8 we map variability types to these levels of variability in the problem or solution space to indicate at what level which variability type would occur. This mapping indicates at what level a variability type would be visible first. It would then be propagated down to the remaining levels. This mapping indicates that all variability types would be visible first at the user expectation, requirements, or architecture/design level (and then be propagated to design, code, etc.). As can be seen in Table 8, variability types occur in either problem or solution space, but not in both at the same time. Also, lower level solution space aspects (code) are not covered by our variability types.

6.1.3. Variability handling mechanisms versus variability realization techniques

Svahnberg et al. proposed a taxonomy of variability realization techniques for software product lines [2]. In their work, variability

Table 7
Comparison of variability types in ESS with variability layers in FORM.

Variability type	Features/layers in FORM			
	Capability	Operating environment	Domain technology	Implementation technique
VT1 – Variability in functionality	✓			
VT2 – Variability in backend systems		✓		
VT3 – Variability in data		✓		
VT4 – Variability in operating systems		✓		
VT5 – Variability in external interfaces		✓		
VT6 – User interface variability				
VT7 – Variability in process	✓			

realization is defined as "ways to implement the variation points for a variant feature". Our definition of handling variability (see Section 1.1) is about describing and resolving variability. Since "ways to implement the variation points for a variant feature" require both describing and resolving variability, it overlaps with Svahnberg et al.'s definition. Thus, in Table 9 we relate variability realization techniques from Svahnberg et al. to our variability handling mechanisms. This mapping indicates what variability realization techniques defined by Svahnberg et al. address similar issues as the variability handling mechanisms that we identified in our study. However, the variability handling mechanisms identified in our study are a bit different to variability realization techniques defined by Svahnberg et al. in the sense that our mechanisms are at a higher abstraction level in the design (with the exception of VM8 – Code switches). Most variability realization techniques in Svahnberg focus on component and implementation structure but omit other aspects of design, such as information, process and deployment. This is in line with our finding related to variability types discussed in Sections 6.1.1 and 6.1.2 that do not relate to lower level variability layers. Therefore, gray rows and columns in Table 9 indicate items from that could not be mapped. Furthermore, some variability handling mechanisms identified in our study can be considered orthogonal to variability realization techniques identified by Svahnberg et al. or even spread across multiple variability realization techniques. For example, VM2 (add-ons) can be supported by several variability realization techniques. Below we explain the mappings in more detail:

- VM1 (enhancement package concept): The enhancement package concept is an overarching mechanism that may be supported by various variability realization techniques. Also, the

Table 8
Mapping of variability types to levels of variability.

Variability type	Variability level according to Gurp et al.				
	User expectations	Requirements	Architecture	Design	Code
VT1 – Variability in functionality	✓				
VT2 – Variability in backend systems			✓		
VT3 – Variability in data		✓			
VT4 – Variability in operating systems			✓		
VT5 – Variability in external interfaces		✓			
VT6 – Variability in user interface				✓	
VT7 – Variability in process	✓				

Table 9
Variability realization techniques versus variability handling mechanisms.

Technique	VM1 – EPC	VM2 – Add-ons	VM3 – Function switching	VM4 – Layering	VM5 – Add-ins	VM6 – Metadata modeling	VM7 – Business rules	VM8 – Code switches
Architecture reorganization								
Variant architecture component								
Optional architecture component		✓			✓			
Binary replacement – linker directives		✓			✓			
Binary replacement – physical		✓						
Infrastructure-centered architecture								
Variant component specializations			✓		✓		✓	
Optional component specializations					✓			
Runtime variant component specializations	✓	✓						
Variant component implementations					✓		✓	
Condition on constant			✓				✓	
Condition on variable			✓				✓	
Code fragment superimposition								✓

enhancement package concept can be used at runtime to provide and select (i.e., activate) several specializations inside a component implementation to satisfy to satisfy the requirements (runtime variant component specializations) [2].

- VM2 (add-ons): Add-ons may require architectural components (extensions) that may or may not be present in a system (optional architecture component). Also, add-ons may be supported by providing the system with alternative implementations of underlying libraries (binary replacement – linker directives). Furthermore, add-ons may require the modification of software after delivery (binary replacement – physical). Also, add-ons mean that implementations are equipped with a number of alternative executions that can be selected at runtime (runtime variant component specializations).
- VM3 (function switching): To facilitate and enable function switching, we may need to support the existence and selection between several specializations inside a component implementation (variant component specializations). Also, it may be necessary to put conditions on constants or on variables (condition on constant, condition on variable). This supports several ways of performing an operation, of which only one will be used at a point in time [2]. In contrast to Svahnberg et al., function switching as defined in our study goes one step further and requires the definition of specializations or conditions in a configuration table.
- VM4 (layering): Layering is a concept to separate concerns or an architectural principle and is therefore not mapped to a particular variability realization technique (gray column in Table 9). As mentioned earlier, some of our mechanisms are at a higher abstraction level in the design.
- VM5 (add-ins): Similar to VM2, add-ins may require architectural components that may or may not be present in a system (optional architecture component), and alternative implementations of underlying libraries (binary replacement – linker directives). Also, we may need to support the existence and selection between several specializations inside a component implementation (variant component specializations). Furthermore, add-ins may mean that component implementations in the product architecture need to be adjusted (variant component implementations) or to include or exclude parts of the behavior of a component implementation (optional component specializations). Again, function switching as found in our study requires that configuration points in the code exist. These configuration points would then facilitate the activities from Svahnberg listed in this paragraph (e.g., by editing configuration points to invoke an architectural component).
- VM6 (metadata modeling): Variability is represented in the data used by a software product, rather than in the software itself. Thus, we could not map it to any variability realization technique (gray column in Table 9). As mentioned earlier, Svahnberg et al. seem to omit some aspects of design, such as information.
- VM7 (business rules): Business rules may require supporting the existence and selection between several specializations inside a component implementation (variant component specializations). Furthermore, different business rules and different flows through a business process may lead to an adjustment of component implementations in the product architecture (variant component implementations). Also, to facilitate and enable different business rules it may be necessary to put conditions on constants or on variables (condition on constant, condition on variable). This supports several ways of performing an operation, of which only one will be used at a point in time [2].
- VM8 (code switches): Code fragment superimposition means that the software is developed to function generically and then allows to superimpose product-specific concerns at a stage when the work with the source code is completed anyway [2]. It is also possible to use code fragment superimposition to introduce variants of other forms that need not have to do with customizing source code to fit a particular product [2]. This idea is supported by code switches found in our study which do not

require implementation of source code at the time of customization, but instead executes pieces of existing source code, depending on the product version.

As can be seen from Table 9, several variability realization techniques from Svahnberg et al. [2] are not directly mapped to our findings and therefore not generalized from our findings (gray rows in Table 9):

- *Architecture reorganization* is about supporting several product-specific architectures by changing the architectural structure of components and their relations in the overall product family architecture to enable changes in control flow and data flow [2]. Even though business rules may require different data and control flows, this is usually not achieved by re-organizing the architecture.
- *Variant architecture component* is about supporting several differing architecture components representing the same conceptual entity [2]. None of our mechanisms is about representing the same conceptual entity.
- *Infrastructure-centered architecture* is about making connections between components a first class entity so that components in the architecture are no longer connected to each other but to the infrastructure [2]. This realization technique could not be found to be related to any variability handling mechanism in our study.

6.2. Implications for researchers

We discuss implications for researchers from three perspectives: First, we elaborate on the validity of our theory (i.e., our concepts in terms of variability types and variability handling mechanisms). Second, we discuss the implications of our study on variability in quality. As found in a recent study [46], variability in quality attributes is not sufficiently addressed by current research. Thus, we reflect on how our findings relate to variability in quality attributes of ESS in practice. Third, we outline potentials for future work for researchers based on our findings.

6.2.1. Evaluation of findings derived from grounded theory

Glaser recommends that findings derived from a grounded theory approach should be evaluated based on four criteria: fit, work, relevance and modifiability [47]. This section only focuses on the evaluation of the theory. General threats to the validity of our case study are discussed in Section 7.

- *Fit* is the ability of categories and their properties to fit realities under study in the eyes of subjects, practitioners and researchers in the area [41]. Thus, a theory fits if it explains the experiences of participants as well as different practitioners who were not involved in theory generation [48]. Since we checked our findings with practitioners not involved in the analysis, our theory seems to be a good fit. However, more data can lead to a refinement of the theory and constructs (e.g., additional variability types or variability handling mechanisms). Thus, researchers may conduct additional case studies in the context of ESS or use our constructs and evaluate them through a broad sample (e.g., through survey research).
- *Work* is the ability of a theory to explain major variations in behavior in the area with respect to the processing of the main concerns of subjects. In our study, the emerging codes and concepts were strongly related to the main concern of ESS, i.e., their ability to be deployed in different contexts.
- *Relevance* is achieved when the criteria of fit and work are met [48]. However, researchers may increase relevance by conducting additional studies in industrial contexts, with sufficient rigor [49].
- *Modifiability* means that a theory is ready for changes to include variations in emergent properties and categories caused by new data [41]. As argued earlier, our set of constructs can be expanded if new data from further studies becomes available.

In addition to these criteria, the ability of a theory to fit and extend previous literature also helps evaluate it. We have discussed how our findings of variability in ESS relate to previous literature in previous sections.

6.2.2. Variability in quality

Variability usually refers to variability in features or functionality. However, variability also affects the quality of software systems [50]. Most current work on variability concerns functional variability while quality attribute variability has not been explored extensively [50,51]. However, similar as with other software system [52–54], quality attributes often drive key architectural decisions when designing variability-intensive ESS [55–57]. Therefore, we tried to understand whether the types of variability identified in our study and the variability handling mechanisms are related to quality attributes of variability-intensive ESS.

- In System 1, quality attributes are a general concern independent of variability. Also, several quality attributes are enforced centrally through product standards (e.g., security, accessibility, translatability, performance, data quality). Quality goals are implicit and exist as overall guidelines or high-level architecture decisions. For System 1, sixteen product standards on different quality-related topics exist. Product standard compliance checks ensure that quality attribute requirements are met.
- In System 2 there is no difference in quality attributes for different business units, except for security. For example, the documentation of a product can be public or private. This security information is captured in the catalog maintenance guidelines for each business unit. Experience with the B2B system over 11 years has shown significant variability in quality attributes. To ensure that quality attributes are met, prototyping before technical design is used, focusing on response time and scalability.

In our study we found that neither the variability types, nor the variability handling mechanisms indicate or address variability in quality attributes. Also, we found no indicator for a special treatment of quality attribute when facilitating variability.

6.2.3. Future work for researchers

In future work, well-known methods from product line engineering could be adopted in action research studies to assess their feasibility in the context of large-scale ESS. For example, approaches could be developed to deal with the complexity of ultra-large scale feature models of such systems. Experiments and case studies could be used to assess how comprehensive such models would be.

Furthermore, as shown in Sections 6.1.1 and 6.1.2, our variability types partially refine existing types. Our variability handling mechanisms focus on higher level variability realization (Section 6.1.3). Thus, from a theory-building perspective, future work can expand theories and concepts to address lower level and or fine-grained variability.

Finally, based on our discussion in Section 6.2.2, studies should be dedicated to investigating variability in quality attributes of ESS. For example, variability in quality attributes can occur due to variations in different market segments or customer profiles (e.g., premium customers may get better performance compared to regular customers). On the other hand, variability in functionality affects

quality. For example, adding additional encryption functionality may affect performance.

6.3. Implications for practitioners

Handling variability in ESS is not easy. The types of variability and the variability handling mechanisms described in this paper should help organizations understand types and related practices when dealing with variability-intensive ESS. The popularity of software product lines has led to several organizations taking on product line practices in non-product line projects. Thus, some organizations may find themselves confused about the role of variability in a non-product line context and may be unsure about the extent to which dedicated variability handling mechanisms should be applied. The variability types and variability handling mechanisms described in this paper should help organizations better understand the meaning of variability and responsibilities for handling different stages of variability. They should also assist organizations in guiding their engineers into explicit variability handling.

As we have shown in the analysis in Section 5 and in Tables 5–8, there is heterogeneity in variability types and variability handling mechanisms. Organizations need to take this heterogeneity into account when considering how to handle variability: variability types have different reasons and origins, appear at different levels of abstraction (e.g., code or design level), are used by different actors (e.g., application engineers or domain engineers), may be used for different purposes and have different focus areas (e.g., defining variability versus resolving variability). Furthermore, variability types are related and do not occur in isolation. In a variability-intensive ESS, different practices can be used to handle different types of variability. While some practices for handling variability may exist in organizations anyway, some may specifically be introduced to handle variability. Others may need to be adjusted for that particular purpose. Furthermore, some practices may be easy to introduce while some may fail.

Furthermore, in Section 1 we described our notion of ESS as applications that are characterized by (a) dealing with large amounts and complex data, (b) dealing with business processes, (c) implementing client–server, distributed multi-user systems rather than local single user systems, (d) being large scale, and (e) being part of a larger existing infrastructure. We found that the list of variability types and variability handling mechanisms reflect this nature of ESS. For example, layering (VM4) as a high level concept to separate concerns is necessary because of the complexity of ESS. Variability in data (VT3) and metadata modeling (VM6) are means to describe and to deal with the large amounts and complex data that ESS must process. Variability in process (VT7) and business rules (VM7) are necessary because ESS deal with business processes. Variability in backend systems (VT2) and variability in operating systems (VT4) and external interfaces (VT5) may be caused by the fact that ESS are often integrated into an existing IT infrastructure.

7. Validity

We classify limitations of this case study as construct validity, external validity and reliability [26,58]. Since the case study is exploratory, internal validity is not a concern as there is no claim about causal relationships [59].

Construct validity: Construct validity is concerned with whether or not the researchers measure what is intended. With this regard, this study is limited as we gathered data only from a limited number of data sources. However, we collected data from two different projects. On the other hand, most collected data concerns System 1 which may pose a threat to the validity of the results concerning System 2. Here, the consistency of our data from different data collection techniques gives us confidence that we have identified valid data. For example, as we have shown in Section 4.3, variability types and mechanisms occurred in two or more data sources (data source triangulation). Moreover, when coding our data, we had to integrate data where different subjects might have used terms and concepts with different meanings [40]. Providing transcripts for comments to participants helped mitigate this problem. Furthermore, we compared our results with theories and findings from similar literature. Finally, we applied member checks by checking with interviewees and incorporating their comments during data analysis.

External validity: External validity is concerned with the statistical generalization of results and to what extent the findings are of interest outside the investigated cases. There is only limited access to organizations that develop large ESS. This makes it difficult to collect enough data points for statistical generalization. Thus, rather than aiming for statistical generalization, we aimed for analytical generalization. This means, our results can be applied to cases that follow our definition of ESS given in Section 1.1 and which have similar characteristics as the cases in our case study (see Section 3.1.1). Finally, comparing our finding with similar literature sharpens generalizability [43]. The presented case study on variability types and variability handling mechanisms is a first of its kind. Further research will help generate a more generalized theory.

Reliability: This aspect is concerned with to how the data analysis depends on the specific researchers. To address this issue, data collection instruments were reviewed, results have been reviewed by case subjects and more than one data collection technique/data source has been used. Also, we found that all data sources provided us with more or less the same findings. This could be an indication of data saturation. The case study involved people with different backgrounds, each expressing their experience and views and providing us with a coherent view on the topic at hand. Nevertheless, we found that the results obtained from the different interviews and document analyses were consistent, which increases our confidence in the trustworthiness of the data. To ensure an audit trial, interviews were recorded and transcribed. Due to privacy concerns of some interviewees, not all interviews could be recorded; however, extensive notes were taken, providing us with a "thick description".

8. Conclusions

In this paper, we studied variability in large-scale enterprise software systems through an industrial case study. We identified seven types of variability and eight mechanisms to handle variability. We outlined future work in Section 6.2. The key lessons we learnt are (a) that some variability types and handling mechanisms are not specific to ESS but occur in software systems in general, (b) some variability types and handling mechanisms appear particularly because of the nature of ESS, (c) that variability types and variability handling mechanisms are highly heterogeneous, with different focus areas and appearances in the software development cycle and d) that our concepts (variability types and variability handling mechanisms) partially refine existing concepts and theories found in literature.

Since the findings were derived from the design of long-lived and commercially successful systems, the findings can be used to form "good practices" of mechanisms on how to deal with variability in large scale enterprise software systems. However, we cannot claim that they are "best practices" since such an absolute statement would require the evaluation of these practices in a broader context. In contrast to "recommended practices" which are usually

understood as published documents that provide technical guidance and preferred procedures to software design, including common terminologies and concepts [60], "good practices" are concrete practices identified based on designing and maintaining two concrete software systems over many years. However, "recommended practices" may also include other new findings from the fields of software architecture and variability management that have been made throughout the life time of those two systems, but which were not used in the design of those systems to achieve variability with e.g., less effort.

Acknowledgments

The authors thank all individuals and organizations that participated in the study. This research is partially sponsored by NWO SaS-LeG, Contract No. 638.000.000.07N07.

Appendix A. Interview guidelines

Interview section	Task
Preparation	Record time and location of interview
Introduction	Explain research and planned activities
	Introduce goal of interview and present interview outline
General information	Record interviewee name, position, background and experience
	Obtain company/project/system information
	Ask general questions about project to understand context
Variability	Ask general questions about variability in the organization and its products
	Explore causes of variability
	Explore issues that cause most problems because of variability
	Explore types of variability
	Ask about the role of the architecture to facilitate variability
	Explore general variability handling
Design process	Explore steps from business process to software design
	Explore about quality attributes in relation to variability
Summary and wrap-up	Summarize interview to interviewee
	Ask for anything else to add or that may be relevant

References

[1] F. Bachmann, P.C. Clements, Variability in Software Product Lines, SEI CMU, Pittsburgh, PA, 2005.

[2] M. Svahnberg, J. van Gurp, J. Bosch, A taxonomy of variability realization techniques, Software – Pract. Ex. 35 (2005) 705–754.

[3] F. van der Linden, K. Schmid, E. Rommes, Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering, Springer Verlag, Berlin/Heidelberg, 2007.

[4] R. Hilliard, On representing variation, in: Workshop on Variability in Software Product Line Architectures, ACM, Copenhagen, Denmark, 2010, pp. 312–315.

[5] J. van Gurp, J. Bosch, Preface, in: J. van Gurp, J. Bosch (Eds.), Software Variability Management Workshop, University of Groningen, Groningen, The Netherlands, 2003, p. 1.

[6] M. Fowler, Patterns of Enterprise Application Architecture, Addison Wesley, Boston, MA, 2002.

[7] S. Thiel, A. Hein, Modeling and using product line variability in automotive systems, IEEE Software 19 (2002) 66–72.

[8] M. Galster, P. Avgeriou, Handling variability in software architecture: problems and implications, in: 9th IEEE/IFIP Working Conference on Software Architecture, IEEE Computer Society, Boulder, CO, 2011, pp. 171–180.

[9] V. Basili, G. Caldiera, D. Rombach, The goal question metric approach, in: J.J. Marciniak (Ed.), Encyclopedia of Software Engineering, John Wiley & Sons, New York, NY, 1994, pp. 528–532.

[10] M. Torchiano, F. Ricca, Six reasons for rejecting an industrial survey paper, in: First International Workshop on Conducting Empirical Studies in Industry (CESI), IEEE Computer Society, San Francisco, 2013, pp. 1–6.

[11] S.H. Chang, S.D. Kim, A variability modeling method for adaptable services in service-oriented computing, in: 11th International Software Product Line Conference, IEEE Computer Society, Kyoto, Japan, 2007, pp. 261–268.

[12] S. Segura, D. Benavides, A. Ruiz-Cortes, P. Trinidad, A taxonomy of variability in web service flows, in: First Workshop on Service-oriented Architectures and Product Lines, SEI, Kyoto, Japan, 2007, pp. 1–5.

[13] M. Sinnema, S. Deelstra, Classifying variability modeling techniques, Inform. Software Technol. 49 (2007) 717–739.

[14] B.G. Glaser, Theoretical Sensitivity: Advances in the Methodology of Grounded Theory, The Sociology Press, 1978.

[15] L. Chen, M.A. Babar, N. Ali, Variability management in software product lines: a systematic review, in: 13th International Software Product Line Conference (SPLC), Carnegie Mellon University, San Francisco, CA, 2009, pp. 81–90.

[16] Y. Ishida, Software product lines approach in enterprise system development, in: 11th International Software Product Line Conference, IEEE Computer Society, Kyoto, Japan, 2007, pp. 44–53.

[17] N. Kozuka, Y. Ishida, Building a product line architecture for variant-rich enterprise applications using a data-oriented approach, in: 1st Workshop on Services, Clouds, and Alternative Design Strategies for Variant-Rich Software Systems ACM, Munich, Germany, 2011, pp. 14.

[18] J. Bayer, O. Flege, P. Knauber, R. Laqua, D. Muthig, K. Schmid, T. Widen, J.-M. DeBaud, PuLSE: a methodology to develop software product lines, in: Symposium on Software Reusability, ACM, Los Angeles, CA, 1999, pp. 122–131.

[19] P. Clements, L. Northrop, Software Product Lines – Practices and Patterns, Addison-Wesley, Boston, MA, 2001.

[20] M. Galster, P. Avgeriou, D. Tofan, Constraints for the design of variability-intensive service-oriented reference architectures – an industrial case study, Inform. Software Technol. 55 (2013) 428–441.

[21] M. Galster, P. Avgeriou, A variability viewpoint for enterprise software systems, in: Joint 10th Working IEEE/IFIP Conference on Software Architecture (WICSA) & 6th European Conference on Software Architecture (ECSA), IEEE Computer Society, Helsinki, Finland, 2012, pp. 267–271.

[22] L. Chen, M.A. Babar, Variability management in software product lines: an investigation of contemporary industrial challenges, in: 14th International Software Product Line Conference, Springer Verlag, Jeju Island, South Korea, 2010, pp. 1–15.

[23] T. Ihme, M. Pikkarainen, S. Teppola, J. Kaariainen, O. Biot, Challenges and Industry Practices for Managing Software Variability in Small and Medium Sized Enterprises, Empir. Sofw. Eng. 19 (2014) 1144–1168.

[24] M. Jaring, J. Bosch, Representing variability in software product lines: a case study, in: Second Software Product Line Conference, Springer Verlag, San Diego, CA, 2002, pp. 15–36.

[25] C. Thoern, Current state and potential of variability management practices in software-intensive smes: results from a regional industrial survey, Inform. Software Technol. 52 (2010) 411–421.

[26] R.K. Yin, Case Study Research – Design and Methods, Sage Publications, London, UK, 2009.

[27] D.E. Gray, Doing Research in the Real World, Sage Publications, London, UK, 2009.

[28] J. Verner, J. Sampson, V. Tosic, N.A.A. Bakar, B. Kitchenham, Guidelines for industrially-based multiple case studies in software engineering, in: Third IEEE International Conference on Research Challenges in Information Science, IEEE Computer Society, Fes, Morocco, 2009, pp. 313–324.

[29] C. Robson, Real World Research: A Resource for Social Scientists and Practitioner-researchers, Blackwell Publishers, Oxford, UK, 2002.

[30] P. Vogt, Dictionary of Statistics and Methodology – A Non-technical Guide for the Social Sciences, Sage Publications, Thousand Oaks, CA, 2005.

[31] P. Runeson, M. Hoest, Guidelines for conducting and reporting case study research in software engineering, Empirical Software Eng. 14 (2009) 131–164.

[32] J. Gerring, Case Study Research – Principles and Practices, Cambridge University Press, Cambridge, NY, 2006.

[33] B.G. Glaser, A. Strauss, The Discovery of Grounded Theory: Strategies for Qualitative Research, Aldine Transaction, Piscataway, NJ, 1967.

[34] C. Urquhart, H. Lehmann, M.D. Myers, Putting the 'Theory' back into grounded theory: guidelines for grounded theory studies in information systems, Inform. Syst. J. 20 (2010) 357–381.

[35] S. Adolph, W. Hall, P. Kruchten, Using grounded theory to study the experience of software development, Empirical Software Eng. 16 (2011) 487–513.

[36] A.C. Strauss, J. Corbin, Basics of Qualitative Research: Grounded Theory Procedures and Techniques, 2nd ed., Sage Publications, Thousand Oaks, CA, 1990.

[37] M.B. Miles, A.M. Huberman, Qualitative Data Analysis, second ed., Sage Publications, Thousand Oaks, CA, 1994.

[38] C.B. Seaman, Qualitative methods in empirical studies of software engineering, IEEE Trans. Software Eng. 25 (1999) 557–572.

[39] K. Krippendorff, Content Analysis: An Introduction to its Methodology, second ed., Sage Publications, Thousand Oaks, CA, 2003.
[40] G.W. Noblit, R.D. Hare, Meta-Ethnography: Synthesizing Qualitative Studies, Sage Publications, Newbury Park, CA, 1988.
[41] B.G. Glaser, Doing Grounded Theory: Issues and Discussions, Sociology Press, 1998.
[42] M. Jaring, J. Bosch, A taxonomy and hierarchy of variability dependencies in software product family engineering, in: 2004 Computer Software and Applications Conference (COMPSAC 2004), Hong Kong, China, vol. 351, 2004, pp. 356–361.
[43] K.M. Eisenhardt, Building theories from case study research, Acad. Manage. Rev. 14 (1989) 532–550.
[44] K.C. Kang, S. Kim, J. Lee, K. Kim, G.J. Kim, E. Shin, FORM: a feature-oriented reuse method with domain-specific reference architecture, Ann. Software Eng. 5 (1998) 143–168.
[45] J. van Gurp, J. Bosch, M. Svahnberg, On the notion of variability in software product lines, in: Working IEEE/IFIP Conference on Software Architecture, IEEE Computer Society, Amsterdam, The Netherlands, 2001, pp. 45–54.
[46] M. Galster, D. Weyns, D. Tofan, B. Michalik, P. Avgeriou, Variability in software systems – a systematic literature review, IEEE Trans. Software Eng. 40 (2014) 282–306.
[47] B.G. Glaser, Basics of Grounded Theory Analysis: Emergence vs, Sociology Press, Forcing, 1992.
[48] R. Hoda, J. Noble, S. Marshall, Self-organizing roles on agile software development teams, IEEE Trans. Software Eng. 39 (2013) 422–444.
[49] M. Ivarsson, T. Gorschek, A method for evaluating rigor and industrial relevance of technology evaluations, Empirical Software Eng. 16 (2011) 365–395.
[50] V. Myllärniemi, M. Raatikainen, T. Mannisto, A systematically conducted literature review: quality attribute variability in software product lines, in: 16th International Software Product Line Conference ACM, Salvador, Brazil, 2012, pp. 41–45.
[51] L. Chen, M.A. Babar, A systematic review of evaluation of variability management approaches in software product lines, Inform. Software Technol. 53 (2011) 344–362.
[52] U. van Heesch, P. Avgeriou, Naive architecting – understanding the reasoning process of students – a descriptive survey, in: 4th European Conference on Software Architecture, Springer Verlag, Copenhagen, Denmark, 2010, pp. 24–37.
[53] U. van Heesch, P. Avgeriou, Mature architecting – a survey about the reasoning process of professional architects, in: 9th Working IEEE/IFIP Conference on Software Architecture, IEEE Computer Society, Boulder, CO, 2011, pp. 260–269.
[54] U. van Heesch, P. Avgeriou, R. Hilliard, Forces on architecture decisions – a viewpoint, in: Joint Working Conference on Software Architecture & 6th European Conference on Software Architecture, IEEE Computer Society, Helsinki, Finland, 2012, pp. 101–110.
[55] S. Balasubramaniam, G.A. Lewis, E. Morris, S. Simanta, D.B. Smith, Challenges for assuring quality of service in a service-oriented environment, in: 2009 ICSE Workshop on Principles of Engineering Service Oriented Systems, IEEE Computer Society, Vancouver, Canada, 2009, pp. 103–106.
[56] E. Johansson, A. Wesslen, L. Bratthall, M. Hoest, The importance of quality requirements in software platform development – a survey, in: 34th Annual Hawaii International Conference on System Sciences, IEEE Computer Society, Honolulu, HI, 2001, pp. 1–10.
[57] S. Mahdavi-Hezavehi, M. Galster, P. Avgeriou, Variability in quality attributes of service-based software systems: a systematic literature review, Inform. Software Technol. 55 (2013) 320–343.
[58] C. Wohlin, M. Hoest, K. Henningsson, Empirical research methods in software engineering, in: R. Conradi, A.I. Wang (Eds.), Empirical Methods and Studies in Software Engineering, Springer Verlag, Berlin/Heidelberg, 2003, pp. 7–23.
[59] S. Easterbrook, J. Singer, M.-A. Storey, D. Damian, Selecting empirical methods for software engineering research, in: F. Shull, J. Singer, D.I.K. Sjoberg (Eds.), Guide to Advanced Empirical Software Engineering, Springer, Berlin, 2008, pp. 285–311.
[60] IEEE Computer Society Software Engineering Standards Committee, IEEE Recommended Practice for Architectural Description of Software-Intensive Systems, 2000.