Variability in Multi-tenant Environments: Architectural Design Patterns from Industry

Jaap Kabbedijk and Slinger Jansen

Utrecht University
Department of Information and Computing Sciences
Princetonplein 5, 3584CC, Utrecht, Netherlands
{j.kabbedijk,s.jansen}@cs.uu.nl

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

Keywords: Software-as-a-Service, Variability, Multi-tenancy, Architectural Patterns.

1 Introduction

Increasingly, product software vendors want to offer their product as a service to their customers [1]. This principle is referred to in literature as Software as a Service (SaaS) [2]. Turning software into a service from a vendor's point of view means separating the possession and ownership of software from its use. Software is still maintained and deployed by the vendor, but used by the customer. The problem of moving a software product from different on-premises locations to one central location, is the fact that it becomes really difficult to comply to specific customer wishes. In order to serve different customers' wishes, variability in a software product is needed to offer specific functionality. By making use of variability in a software product, it is possible to supply software functionality as optional modules, that can be added to the product at runtime. Applying this principle can overcome many current limitations concerning software use, deployment, maintenance and evolution in a SaaS context [3]. It also reduces support costs, as only a single instance of the software has to be maintained [4].

Besides complying to specific customer requirements, a software vendor should be able to offer a service to a large number of customers, each with their own

O. De Troyer et al. (Eds.): ER 2011 Workshops, LNCS 6999, pp. 151–160, 2011.

requirement wishes, without running into scalability and configuration problems [5]. The solution to this problem is the use of multi-tenancy within a SaaS product. Multi-tenancy can be seen as an architectural design pattern in which a single instance of a software product is run on the software vendors infrastructure, and multiple tenants access the same instance [6]. It is one of the key competencies to achieve higher profit margins by leveraging the economy of scale [7]. In contrast to a model incorporating multiple users, multi-tenancy requires customizing the single instance according to the varying requirements among many customers [8]. Currently, no well documented techniques are available on how to realize the variability needed in multi-tenant SaaS environments.

First the research method is discussed in section 2, after which the most important concepts in this paper will be explained and discussed in section 3. Then, the trade-off between the level of variability needed and the number of customers is discussed in section 4, followed by three architectural design patterns for variability in SaaS product in section 5. The paper ends with a conclusion and future research in section 6.

2 Research Method

In this research, the variability realization techniques (VRTs) currently described in literature and in place in large SaaS providers are observed. In order to do this, a thorough literature study has been performed, in which the combinations of *variability, saas* and *variability, multi-tenancy* were used as keywords in Google Scholar[1]. The VRTs discussed in the papers having the previous mentioned keywords in their title or abstract were collected and patterns in those patterns were put in a pattern database. A total of *27* papers was collected this way. Besides the literature study, two independent case studies were performed at large ERP providers who recently launched their enterprise resource planning (ERP) software as a service through the Internet (refered to as ErpCompA and ErpCompB from here on). ErpCompA has a turnover of around 250 million euros and around 20,000 users using their online product, while ErpCompB has a turnover of around 50 million euros and around 10,000 users. The case studies were performed using the case study research approach by Yin [9].

The VRTs are presented as architectural design patterns and created based on the Design Science principles of Hevner [10], in which a constant design cycle consisting of the construction and evaluation of the VRTs takes place. The initial model is constructed using a exploratory focus group (EFG) [11], consisting out of participants from academia and the case companies, and a systematic literature review [12]. The focus group has been carefully selected and all participants have experience in the area of variable multi-tenant SaaS-environments. Additional validation of the VRTs was done conducting interviews with software architects within the two case companies [13].

[1] Google Scholar (www.scholar.google.com indexes and searches almost all academic publishers and repositories world-wide.

3 Related Work and Definitions

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

Multi-tenancy
Multi-tenancy can be defined as the ability to let different tenants "share the same hardware resources, by offering them one shared application and database instance, while allowing them to configure the application to fit their needs as if it runs on a dedicated environment" [5]. A tenant refers to an organization or part of an organization with their own specific requirements, renting the software product. We define different levels of multi-tenancy:

- **Data Model Multi-tenancy:** All tenants share the same database. All data is typically provided with a tenant specific GUID in order to keep all data separate. Even better is native support for multi-tenancy in the database management system [14].
- **Application Multi-tenancy:** Besides sharing the same database, all tenants also share the same instance of the software product. In practice, this could also mean a couple of duplications of the same instance, coupled together with a tenant load balancer [8].
- **Full Multi-tenancy:** All tenants share the same database and software instances. They can also have their own variant of the product, based on their tenant requirements. This level of multi-tenancy adds variability to the software product.

All items above are sorted on ascending implementation complexity.

Variability
The concept of variability comes from the car industry, in which different combinations of for example chassis, engine and color were defined as different *variants*. In software the concept is first introduced in the area of software product lines [15], in which variability is defined as "the ability of a software system or artefact to be efficiently extended, changed, customized or configured for use in a particular context" [16]. Within the area of software product lines, software is developed by the software vendor and then shipped to the customer to be run on-premises. This means variants have to be compiled before product shipping. Within the area of Software-as-a-Service, software is still developed by the software vendor, but the product is served to all customers through the internet from one central place [3,8]. In principle, all variants can be composed the moment customers ask for some specific functionality, so at run-time.

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

- **Segment Variability:** Product variability based on the segment a tenant is part of. Examples of such variability issues are different standard currencies

or tax rules per country or a different layout for SMEs and sole proprietorships.
- **Tenant-oriented Variability:** Product variability based on the specific requirements of a tenant. Examples of such variability issues are different background colors or specific functionality.

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

- **Low: Look and Feel:** Changes only influencing the visual representation of the product. These changes only occur in the presentation tier (tier-based architecture [17]) or view element (MVC-base architecture [18]). Examples include different background colors or different element sorting in lists.
- **Medium: Feature:** Changes influencing the logic tier in tier-based architecture or the model or controller element in a MVC-based architecture. Examples include the changes in workflow or the addition of specific functionality.
- **High: Full:** Variability of this level can influence multiple tiers at the same time and can be specific. Examples of this level of variability includes the ability for tenant to run their own program code.

The scope of this research is focussed on *runtime tenant-oriented low and medium variability in multi-tenant SaaS deployments.*

Design Patterns

The concept of patterns was first introduced by Christopher Alexander in his book about the architecture of towns [19]. This concept was quickly picked up in the software engineering world and led to the famous 'Gang of Four' pattern book by Gamma et al. [20]. This book describes several patterns that are still used today and does this in a way that inspired a lot of subsequent pattern authors. The definition of a pattern used in this paper originates from the Pattern Oriented Software Architecture series [21,22,23,24,25] and reads: "A pattern for software architecture describes a particular recurring design problem that arises in specific design contexts, and presents a well-proven generic scheme for its solution. The solution scheme is specified by describing its constituent components, their responsibilities and relationships, and the ways in which they collaborate."

Patterns are not artificially created artifacts, but evolve from best practices and experiences. The patterns described in this paper result from several case studies and discussions with experienced software architects. All patterns have proven to be a suitable solution for the problems described in section 5, since they are applied in successful SaaS products at our case companies. Also, all patterns are described language or platform independent, so the solution can be applied in various situations in the Software-as-a-Service domain. More information on future research concerning the patterns proposed can be found in section 6.

4 User-Variability Trade-off

The best solution for deploying a software product from a software vendor's perspective depends on level of resources shared and the level of variability needed to keep all users satisfied. In figure 1 four deployment solutions are introduced, that are considered best practices in the specific situations shown. In this section, the need for multi-tenant deployment models is explained.

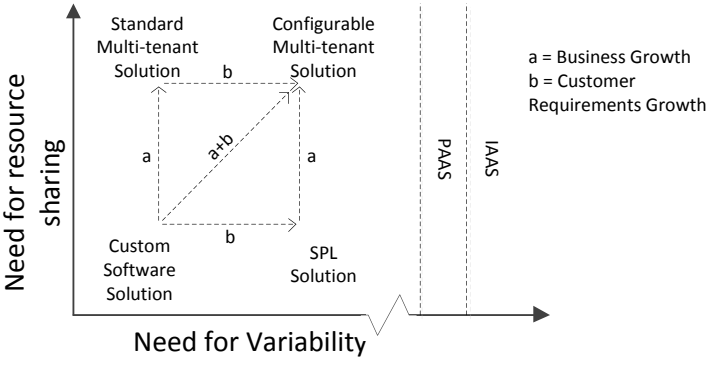

Fig. 1. Level of variability versus Number of users

By using the model shown in figure 1, software vendors can determine the best suited software deployment option. On the horizontal axis, the need for variability in a software product is depicted and the number of customers is shown on the vertical axis. For a small software vendor who does not have a lot of customers with specific wishes, a standard *custom software solution* is sufficient. The more customers software vendors get (business growth), the higher the need for a *standard multi-tenant solution* because of the advantages in maintenance. When the amount of specific customer wishes grows, software vendors can choose the software product line (SPL) approach to create variant for all customers having specific requirements. This solution can lead to a lot of extra maintenance issues as the number of customers grows. In case of a large number of customers having specific requirements, a *configurable multi-tenant solution* is the best solution for software vendors, keeping an eye on performance and maintenance.

5 Variability Patterns

In this section three patterns are described that were observed in the case studies that were conducted. The patterns are designed based on patterns observed within the case companies' software product, extended by patterns already documented in literature [20,16]. All patterns will be explained by an UML-diagram, together with descriptive topics proposed by Buschmann et al. [24] and Gamma et al. [20]

5.1 Customizable Data Views

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

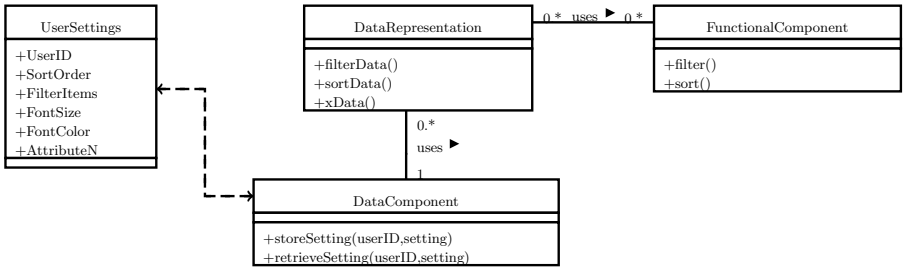

Fig. 2. Customizable Data Views Pattern

Intent - To give the tenant the ability to indicate and save his preferences on the representation of data shown.
Motivation - In a multi-tenant solution it is important to give tenants the feeling they can customize the product the way they want it. This customizability is most relevant in parts of the product where data is presented to the tenant.
Solution - In this variability pattern (cf. figure 2), the representation of data is performed at client side. Tenants can for example choose how they want to sort or filter their data, while the data-queries do not have to be adapted. The only change needed to a software product is the introduction of tenant-specific representation settings. In this table, all preferred font colors, sizes and sort option can be stored in order to retrieve this information on other occasions to display the data again, according to the tenant's wishes.
Explanation - As can be seen in the UML representation of the pattern in figure 2, the *DataRepresentation* class can manipulate the appearance of all data by making use of a *FunctionalComponent* able of sorting, filtering, etcetera. All settings are later stored by a *DataComponent* in a specific *UserSettings* table. Settings can later be retrieved by the same DataComponent, to be used again by the DataRepresentation class and FunctionalModule.
Consequences - By implementing this pattern, one extra table has to be implemented. Nothing changes in the way data selection queries have to be formatted. Representation of all data has to be formatted in a default way, except if a tenant changes this default way and stores his own preferences.
Example - In a bookkeeping program, a tenant for example, can decide what columns he wants to display and how he wants to order them. By clicking the columns he wants to display, his preferences are saved in the database. When the tenant uses the product again later, his preferences are fetched from the database and applied to his data.

5.2 Module Dependent Menu

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

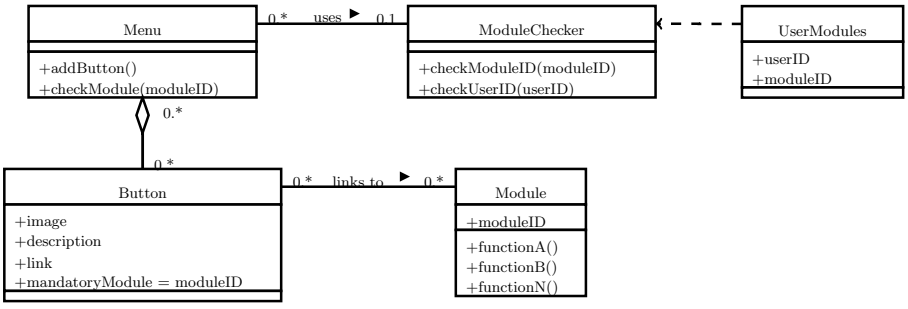

Fig. 3. Module Dependent Menu

Intent - To provide a custom menu to all tenants, only containing links to the functionality relevant to the tenant.
Motivation - Since all tenants have specific requirements to a software product, they can all use different sets of functionality. Displaying all possible functionality in the menu would decrease the user experience of tenants, so menus have to display only the functionality that is relevant to the tenant.
Solution - The pattern proposed (cf. figure 3), creates a menu out of different buttons based on the modules associated to the tenant. Every time a tenant displays the menu, the menu is built dynamically based on the modules he has selected or bought.
Explanation - The *Menu* class aggregates and displays different *buttons*, containing a link a specific module and the prerequisite for displaying this link (*mandatoryModule*). The selection of buttons is done, based on the results of the *ModuleChecker*. This class checks whether an entry is available in the *UserModules* table, containing both the ID of the tenant (user) and the mandatory module. If an entry is present, the *Menu* aggregates and displays the button corresponding to this module.
Consequences - To be able to use this pattern, an extra table containing user IDs and the modules available to this user has to be implemented. Also, the extra class *ModuleChecker* has to be implemented. All buttons do need a notion of a mandatory module that can be checked by the ModuleChecker to verify if a tenant wants or can have a link to the specific functionality.
Example - In a large bookkeeping product, containing several modules that can be bought by a tenant, the menus presented to the tenant can be dynamically composed based on the tenant's license.‘

5.3 Pre/Post Update Hooks

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

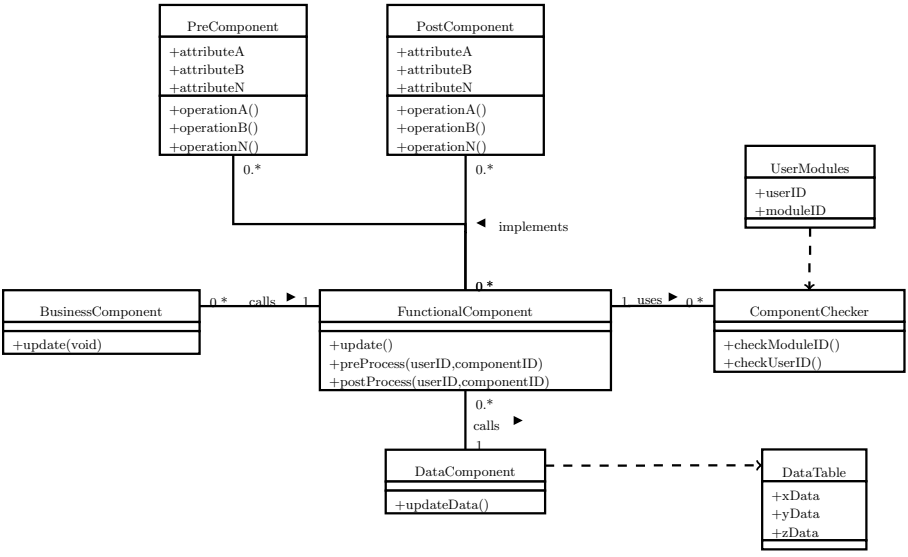

Fig. 4. Pre- Post Update Hooks

Intent - To provide the possibility for tenants to have custom functionality just before or after an event.
Motivation - In business oriented software, workflows often differ per tenant. To let the software product fit the tenants business processes best, extra actions could be made available to tentants before or after an event is called.
Solution - The pattern introduced here (cf. figure 4), makes use of a component able of calling other components before and after the update of data. The tenant-specific modules are listed in a separate table, similar to the pattern described in section 5.2.
Explanation - Before the *FunctionalComponent* calls the *BusinessComponent* in order to perform an update, the *ComponentChecker* is used to check the *UserModules* table if a tenant wants and may implements an extra component before the update is performed. After this, the *BusinessComponent* is called and the update is performed. The *DataComponent* takes care of the writing of data to a specific data table. After this, the *ComponentChecker* again checks the *UserModules* table and a possible *PostComponent* is called.
Consequences - Extra optional components have to be available in the software system in order to be able to implement this pattern. The amount and type of components available depends on the tenants' requirements.

Example - In a bookkeeping program, tenants can choose, whether they want to update a third party service as well by using a component that uses the API of a third party service to make changes there. If so, the FunctionalComponent can call the third party communicator after an internal update is requested.

6 Conclusion and Future Research

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

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

Acknowledgment. This research is part of the Product as a Service project. Thanks goes to both case companies and their helpful architects, experts and developers.

References

1. Ma, D.: The Business Model of "Software-As-A-Service". In: IEEE International Conference on Services Computing, SCC 2007, pp. 701–702. IEEE, Los Alamitos (2007)
2. Gold, N., Mohan, A., Knight, C., Munro, M.: Understanding service-oriented software. IEEE Software 21(2), 71–77 (2005)
3. Turner, M., Budgen, D., Brereton, P.: Turning software into a service. Computer 36(10), 38–44 (2003)
4. Dubey, A., Wagle, D.: Delivering software as a service. The McKinsey Quarterly 6, 1–12 (2007)
5. Bezemer, C., Zaidman, A.: Multi-tenant SaaS applications: maintenance dream or nightmare? In: Proceedings of the International Workshop on Principles of Software Evolution (IWPSE), pp. 88–92. ACM, New York (2010)
6. Bezemer, C., Zaidman, A., Platzbeecker, B., Hurkmans, T., Hart, A.: Enabling multi-tenancy: An industrial experience report. In: 26th IEEE Int. Conf. on Software Maintenance, ICSM (2010)

7. Guo, C., Sun, W., Huang, Y., Wang, Z., Gao, B.: A framework for native multi-tenancy application development and management. In: The 9th IEEE International Conference on E-Commerce Technology, pp. 551–558 (2007)
8. Kwok, T., Nguyen, T., Lam, L.: A software as a service with multi-tenancy support for an electronic contract management application. In: IEEE International Conference on Services Computing, SCC 2008, vol. 2, pp. 179–186. IEEE, Los Alamitos (2008)
9. Yin, R.: Case study research: Design and methods. Sage Publications, Inc., Thousand Oaks (2009)
10. Hevner, A.R., March, S., Park, J., Ram, S.: Design science in information systems research. Mis Quarterly 28(1), 75–105 (2004)
11. Tremblay, M.C., Hevner, A.R., Berndt, D.J.: The Use of Focus Groups in Design Science Research. Design Research in Information Systems 22, 121–143 (2010)
12. Cooper, H.: Synthesizing research: A guide for literature reviews (1998)
13. Runeson, P., Höst, M.: Guidelines for conducting and reporting case study research in software engineering. Empirical Software Engineering 14(2), 131–164 (2009)
14. Schiller, O., Schiller, B., Brodt, A., Mitschang, B.: Native support of multi-tenancy in RDBMS for software as a service. In: Proceedings of the 14th International Conference on Extending Database Technology, pp. 117–128. ACM, New York (2011)
15. Pohl, K., Böckle, G., van der Linden, F.: Software product line engineering: foundations, principles, and techniques. Springer-Verlag New York Inc., Secaucus (2005)
16. Svahnberg, M., van Gurp, J., Bosch, J.: A taxonomy of variability realization techniques. Software: Practice and Experience 35(8), 705–754 (2005)
17. Eckerson, W.: Three Tier Client/Server Architectures: Achieving Scalability, Performance, and Efficiency in Client/Server Applications. Open Information Systems 3(20), 46–50 (1995)
18. Krasner, G., Pope, S.: A description of the model-view-controller user interface paradigm in the smalltalk-80 system. Journal of Object Oriented Programming 1(3), 26–49 (1988)
19. Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., Angel, S.: A pattern language. Oxford Univ. Pr., Oxford (1977)
20. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software, vol. 206
21. Buschmann, F.: Pattern-Oriented Software Architecture: A System of Patterns, vol. 1. John Wiley & Sons, Chichester (1996)
22. Schmidt, D.: Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, vol. 2. Wiley, Chichester (2000)
23. Kircher, M., Jain, P.: Pattern-Oriented Software Architecture : Patterns for Resource Management, vol. 3 (2004)
24. Buschmann, F., Henney, K., Schmidt, D.: Pattern-Oriented Software Architecture: Pattern Language for Distributed Computing, vol. 4. Wiley, Chichester (2007)
25. Buschmann, F.: Pattern-Oriented Software Architecture : On patterns and Pattern Languages, vol. 5 (2007)