The Role of Variability Patterns in Multi-tenant Business Software

(Invited Paper)

J. Kabbedijk, S. Jansen
Information and Computing Sciences
Utrecht University, Utrecht, Netherlands
{*J.Kabbedijk,S.Jansen*}*@uu.nl*

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

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

***Keywords*-multi-tenancy; variability; software patterns; SaaS; online business software**

I. Introduction

Within business software a frequently studied shift can be observed [1] in which software products are no longer delivered to customers and deployed on-site *(on-premises)*, but are deployed at a central location and offered to customers online *(Software-as-a-Service; SaaS)*. Using the on-premises deployment method, one instance (i.e. one running copy of the software) of a software product is used by one customer. The product can be *tailored* or *customized* to comply to specific customer requirements in case the standard product functionality does not align with the requirements needed by the customer in order to support the business processes in place [2]. Another way to satisfy specific customer requirements is to create different products, based on a software product core containing all general requirements shared by all customers, which are generated in a software product line [3]. The ability to create different software products based on one software product core is referred to as variability and is defined as "the ability of a software system of artefact to be efficiently extended, changed, customized or configured for use in a particular context" [4].

Software variability was first studied in software product lines. An attempt to give a complete taxonomy of variability in software products was done by Svahnberg, van Gurp and Bosch [4], but this overview is focussed on software product lines and omits to take online software products into account. Traditionally, variability only takes on-premises software into account, which leads the fact software products containing specific features have to be created before shipping the software (i.e.early binding time). Online software however, only profits from *run-time* binding times since it would be undesirable to restart or redeploy a software product whenever changes are made (as would be the case with design-time binding for example). Within online software products, variability is the result of the *configurability* of a product. The higher the configurability of a product is, the more variable a product is. A variable product aims to provide customers with "a multitude of options and variations using a single code base, such that it is possible for each tenant to have a unique software configuration" [5].

Using the SaaS deployment model, it is still possible to have one instance of a software product per customer *(Application Server Provider Model; ASP)* [6], but because of the fact the software instance is now deployed in a central location, multiple customers can potentially use the same instance. Sharing one instance of a software product with multiple customers, from a software vendor point of view, can be preferred above having separate instances because of (among others) economies of scale, easier data sharing, lower maintenance costs and improved scalability. Sharing a software instance with multiple customers however, makes it impossible to have customized software products for a specific customer. In other words, Multi-Tenant (MT) software should be able to fulfil all different customer requirements, while still profiting from shared resources. Whenever multiple customers share one instance of a software product and are able to configure the application to meet their requirements, we refer to the application as a *multi-tenant* SaaS application [7]. Tenants are organizational entities (customers) using the application and usually consisting of multiple users. These tenants should be able to configure the product in the way they want because of this, the product should have the right level of variability at the right places.

Question remains however how to implement this variability in an efficient way. Within software architecture, specific problems are often solved in the same way. If the

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

II. Concepts

A. Tenant-based Variability

As indicated in the previous section, there is a need for variability in multi-tenant business software products. An important difference is made in variability management between *internal* and *external* variability [3]. The difference between these two variability types is crucial for defining tenant-based variability and relating it to the current variability concepts.

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

The concept of external run-time variability is closely related to tenant-based variability, but since external runtime variability is related to customers in a *product line* context, it is not geared towards online (SaaS) software. The fact that in multi-tenant SaaS products, all customers share one software instance, causes changes in the standard way of thought related to external variability and adapt it to the online domain. An effort to use the concept of external runtime variability in a SaaS context is done by Mietzner, Metzger, Leymann and Pohl [8], who refer to this type of variability as *Customer-driven variability*. The concept of *tenant-based variability* used in this research is closely related to customer-driven variability, with an emphasis on multi-tenant software products. Different customers have different requirements to a software product. These differences require that the online software product is configurable to allow for the varying requirements.

B. Variability Patterns

Tenant-based variability in a software product can be implemented in several different ways. The appropriate solution depends heavily on the exact problem that needs to be solved. A good way of solving a specific problem is to apply a pattern related to this problem. Patterns are defined as "a general reusable solution to a frequently occurring problem in a certain context" by Gamma, Helm, Johnson and Vlissides [9] in their leading design patterns book, but there is more to software patterns that makes them particularly useful for our research. A common mistake related to software patterns is the thought that they are a highly technical representation of a certain design choice, only readable and understandable by programmers. Patterns are far more general than that and can be documented in a few different ways. Fowler listed a few common pattern description forms [10] like for example *Alexandrian*, *GoF* or *POSA* style. A combination of the different description forms can be used depending on the specific patterns. The forms are not prescriptive and should be used as guidelines in writing a pattern instead of set-in-stone rules.

III. Conceptual Model

A. Model Explanation

This section shows the relationships between the different concepts that play an important role within this research domain. Figure 1 shows all concepts as squares, connected by lines indicating the relationships. The left of the figure shows the problem domain, while the right of the figure shows the attributes of a software pattern that can be used to assess the problem. In the *Software Product Domain*, the *MT Software Product (MTSP)* plays a central role. The MTSP is used by different *tenants*, all having their own specific *requirements*. The MTSP contains certain functionality, indicated as *features* in figure 1. Features can be *fixed*, like for example the functionality of a software product to display financial data in a spreadsheet. Features can also be *variable* and influenced by the specific preferences of a tenant. Variable features could be the possibility to adapt the workflow within the MTSP or the ability to add specific data entities.

The *Software Pattern* contains a *problem* occurring in a specific *context*. The problem and context are related to the software product domain, since this is the area of study. The problem and context in the domain of tenant-based variability will always have to do with tenants having specific requirements and because of this the need for a variable multi-tenant online business software product. A specific problem always exists of different drivers that are the cause of the problem. These drivers are defined as *forces* within a pattern context [11]. By making all the forces that are part of a problem explicit, and placing the problem in a properly defined context, a specific and useful *solution* can be documented. The implementation of a solution always has certain *consequences*, either advantages or liabilities [12].

The conceptual model presented in figure 1 identifies the most important concepts and related to multi-tenant

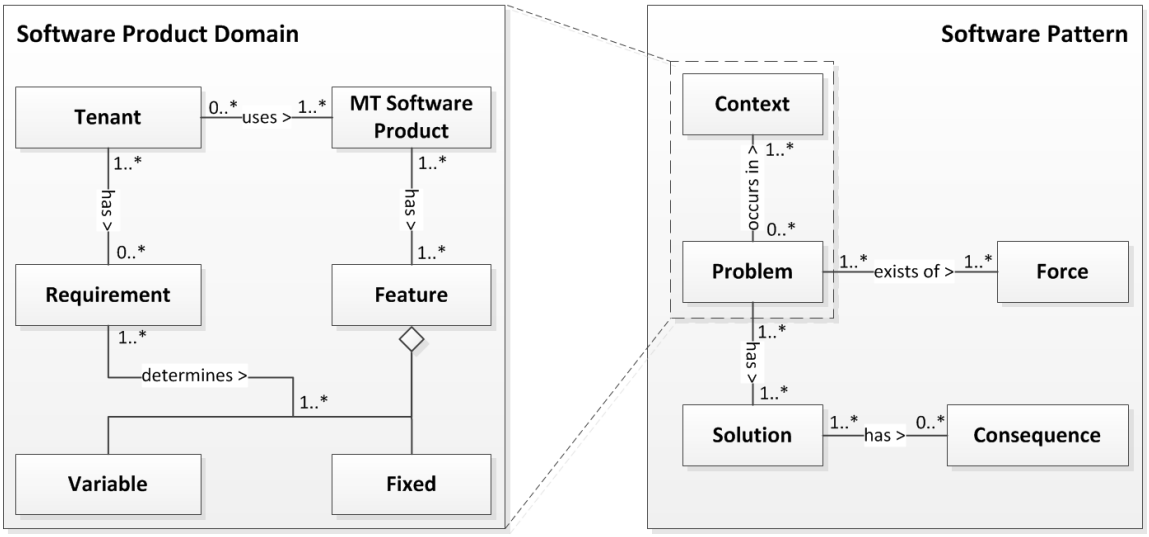

Figure 1. Conceptual Model: The role of variability patterns in multi-tenant business software

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

B. Application Example

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

The identified *problem* in this *context* is caused by the fact customers have different business processes (*force*) and only one instance of the software is deployed on the server that is used by all tenants (*another force*). The *solution* that is applied by the MTSP vendor is the use of a component in the software product, capable of calling tenant-dependent components just before or after a certain action in the workflow is performed. By doing this, the workflow within the MTSP becomes variable in such a way that different tenants can have different actions performed by the MTSP at certain time (*consequence*). The solution however also causes the MTSP to contain additional tables listing the tenant-dependent modules and checking for possible modules before and after each step in the workflow. This may cause performance issues if system load is high (*another consequence*). The make this pattern complete, a name (e.g. pre/post update hooks) and a clear diagram of the solution has to be added, but since this only a illustrative example this has been omitted for the sake of brevity.

IV. Discussion

There is more to patterns than only the definition given by Gamma et al. [9] on patterns being “a solution”. Besides being a solution, patterns also need to have a certain **“goodness”** and **recurrence** before they can be considered a pattern. The recurrence of a solution can be shown by performing a large number of case studies and reporting on the times the specific pattern is observed. A high recurrence however, does not necessarily make a solution a pattern. Whenever a solution is observed frequently, but the solution has serious liabilities, the solution could even be considered an *anti pattern* [13].

Assessing the goodness of a pattern is difficult. Whether a solution is good or bad depends fully on the context of the problem. Scalability may for example be no issue at all for a software product aimed at five customers, while it is of the highest importance for a large ERP product aimed at thousands of SMEs. A way to still say something about the goodness of a pattern is to be thorough in identifying all the forces playing a role within the problem domain. If the forces are clearly listed, the consequences of applying the pattern can be related to the forces. The better all forces are handled, the higher the probability the

pattern is a good solution. Possible liabilities should be analysed and, whenever possible, solutions for mitigating the liabilities should be given. Because of the generalized, implementation independent character of a software pattern, the goodness of a pattern can never be *validated*. However, by being complete in identifying forces and consequences, the goodness of a pattern can be evaluated.

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

V. Conclusion

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

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

Acknowledgment

This research is funded by the NWO/ICT-Regie 'Product as a Service' project.

References

[1] A. D'souza, J. Kabbedijk, D. Seo, S. Jansen, and B. S., "Software-as-a-service: Implications for business and technology in product software companies," in *16th Pacific Asia Conference on Information Systems (PACIS)*, Ho Chi Minh City, Vietnam, 2012 (in press).

[2] W. Sun, X. Zhang, C. Guo, P. Sun, and H. Su, "Software as a service: Configuration and customization perspectives," in *Congress on Services Part II*. IEEE, 2008, pp. 18–25.

[3] K. Pohl, G. Böckle, and F. van der Linden, *Software product line engineering: foundations, principles, and techniques*. Springer-Verlag New York Inc, 2005.

[4] M. Svahnberg, J. van Gurp, and J. Bosch, "A taxonomy of variability realization techniques," *Software: Practice and Experience*, vol. 35, no. 8, pp. 705–754, 2005.

[5] P. Arya, V. Venkatesakumar, and S. Palaniswami, "Configurability in saas for an electronic contract management application," in *Proceedings of the 12th international conference on Networking, VLSI and signal processing*. World Scientific and Engineering Academy and Society (WSEAS), 2010, pp. 210–216.

[6] L. Tao, "Shifting paradigms with the application service provider model," *Computer*, vol. 34, no. 10, pp. 32–39, 2001.

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

[8] R. Mietzner, A. Metzger, F. Leymann, and K. Pohl, "Variability modeling to support customization and deployment of multi-tenant-aware Software as a Service applications," in *Proceedings of the 2009 ICSE Workshop on Principles of Engineering Service Oriented Systems*. IEEE Computer Society, 2009, pp. 18–25.

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

[10] M. Fowler, *Analysis Patterns: reusable object models*. Addison-Wesley, 1997.

[11] F. Buschmann, K. Henney, and D. Schmidt, *Pattern-oriented software architecture: On patterns and pattern languages*. John Wiley & Sons Inc, 2007.

[12] D. Schmidt, "Using design patterns to develop reusable object-oriented communication software," *Communications of the ACM*, vol. 38, no. 10, pp. 65–74, 1995.

[13] W. Brown, R. Malveau, and T. Mowbray, "Antipatterns: refactoring software, architectures, and projects in crisis," 1998.