A Pattern for Enabling Multitenancy in Legacy Application

Flavio Corradini, Francesco De Angelis, Andrea Polini and Samuele Sabbatini
University of Camerino, via del Bastione 2, Camerino, Italy

Keywords: Multi-tenancy, Cloud Migration, Cloud Pattern.

Abstract: Multitenancy is one the new property of cloud computing paradigm that change the way of develop software. This concept consists in the aggregation of different tenant in one single istance in contrast with the classic single-tenant concept. The aim of multitenancy is the reduction of costs, the hardware needed is less than single-tenant application, and also the mantainance of the system is less expensive. On the other hand, applications need an high configuration level in order to satisfy the requirements of each tenant. In this paper is presented a pattern that enable legacy applications to handle a multitenancy database. After the presentation of the different approach that implements multitenancy at database system, it is proposed the pattern that aims to interact with this kind of database managing the different customization of different tenant at database level.

1 INTRODUCTION

The concept of cloud computing is exploding during last year. Although the concept of utility computing was introduced about fifty years ago (Parkhill, 1966), it began to be a commercial need only in the early 2000s. The fact that this new paradigm is driven by commercial aspects and not from a real scientific study has led to the creation of different definitions (Vaquero et al., 2008) depending on the commercial context. NIST (Mell and Grance, 2011) provides the most used definition of cloud. They defines the cloud model as a composition of five essential characteristics, three service models, and four deployment models. The concept is totally revolutionary in software development as foundries have been in the hardware industry (Armbrust et al., 2009). One of the main challenges is the ability to migrate legacy application developed with previous methodologies into a new environment and making them cloud compliant. This challenge is due to the fact that legacy application have been implemented with previous methods without taking into considerations concepts unknown until the advent of cloud (i.e. elasticity, scalability and multi-tenancy).

This work focuses its attention on multi-tenancy concept, in particular on how to adapt legacy application that already exists in order to take advantage from this new concept. The multi-tenancy, within the software architecture community, is usually referred to as the ability to serve multiple client organizations through one instance of a software product and it can be seen as an high level architectural pattern in which a single instance of a software product is hosted on the software vendor's infrastructure, and multiple customers access the same instance (Bezemer and Zaidman, 2010b).

Before going in deep with the multi-tenancy, it is important to underline the difference between multi-tenant and multi-user applications. In a multi-user application we assume that all users are using the same application with limited configuration options. In a multi-tenant application, we assume that each tenant has the possibility to configure the application heavily. This results in the situation that, although tenants are using the same building blocks in their configuration, the appearance or work flow of the application may be completely different for two tenants (Bezemer and Zaidman, 2010a). In order to understand the meaning of multi-tenancy, it can be used the definition formulated in (Kabbedijk et al., 2015) after an evaluation through a Systematic Literature Review. They analysed the definition of 43 different resources extrapolating a definition using the most common word founded. The result of this process is this definition:

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

Corradini, F., Angelis, F., Polini, A. and Sabbatini, S.
A Pattern for Enabling Multitenancy in Legacy Application.
In *Proceedings of the 6th International Conference on Cloud Computing and Services Science (CLOSER 2016) - Volume 2*, pages 257-264
ISBN: 978-989-758-182-3

Multi-tenancy concept involves the entire software stack of applications. In this regard, it is necessary to modify the entire stack to adapt legacy applications to the new model, focusing the attention on the database and application levels. There are several advantages related to multi-tenancy. Here are the principal ones:

- The possibility to share hardware resources, enabling cost reductions (Wang et al., 2008);
- A high degree of configurability, enabling each customer to create his own look-and-feel and workflow within the application (Jansen et al., 2010);
- A shared application and database instance, enabling easier maintenance (Kwok et al., 2008).

The aim of the study is to propose a pattern that can enables applications to manage a multi-tenancy database. Despite the management of a relational database has been defined by several studies, the application adaptation to this new data management model is still in process. This research is related to the Open City Platform project (OCP project, SCN_00467) founded by the Italian Ministry (Ministero dell'Istruzione, dell'Universita e della Ricerca) in the Smart Cities and Communities and Social Innovation initiative (OCP Consortium, 2015). This project aims to migrate the applications used by some Public Administration in an private cloud infrastructure. In this context, this pattern helps companies that are working in the migration of their application in the OCP Cloud Platform.

This paper is structured as follows: Section 2 presents some related works. Section 3 shows the approach that can be used to create a multi-tenancy database. Section 4 proposes a pattern that could enable multi-tenancy in legacy application. Moreover, in that section it is presented the survey that validate the pattern proposed. Section 5 focuses on conclusion and future works.

2 RELATED WORK

This research is part of Italian national project that aims to migrate legacy application in a cloud platform. The aim of this project is very close to other two European project: ARTIST(Artist Consortium, 2015) and REMICS (Remics, 2015). ARTIST and REMICS are two projects very closed to the aim of the research herein. These projects are funded by the European Community, and they focus their aim on migration using Model Driven Engineering (Object Management Group, 2015b). Both projects aim to develop different tools of different part of the migration. REMICS ended in the 2013 and it focused the attention on the recovery, migration, validation and supervising processes of the migration itself. However this project did not cover challenges such as elasticity, multi-tenancy and other non-functional properties. ARTIST focuses on migrating legacy software written in Java and C. The project is still open and it tries to support the migration in every aspect.

This paper focuses its attention on multi-tenancy. This property is very important with the advent of the cloud computing. Several researches are working in the direction of defining an approach to develop cloud native applications or adapt legacy applications in order to enable multi-tenancy. Bezemer et. al (Bezemer and Zaidman, 2010a) propose an architectural pattern that aims to enable multi-tenancy in single-tenant application. In order to solve the problem, the solution proposes three additional layers to be included in the application. The first layer is the Authentication layer that is used to manage different tenants. The second one is the Configuration layer that is used to highly configure applications. The last layer regards the interface between the application and database layer that help to adapt the query. The validation of this approach is proposed in (Bezemer et al., 2010) where the authors validated it with an Industrial Experience Report. This works is very close to our work but they offer an high level point of view that aim to cover all the aspects of the multi-tenancy.

In (Kang et al., 2011) Kang et al. propose a conceptual architecture of a SaaS platform that enables the execution of configurable and multi-tenant SaaS application. The platform allows to configure five aspects of SaaS software (User Interface, Data Model, Organizational Structure, Workflow, Business Logic). In addition, meta-data driven architecture is applied for providing multi-tenancy of SaaS application.

Another important work in multi-tenancy is (Mietzner et al., 2009). In that work, it was described how multi-tenancy can be achieved introducing and evaluating a set of patterns that can be used to design, develop and deploy process-aware service-oriented SaaS applications.

The study of Kabbedijk et. al (Kabbedijk and Jansen, 2011) is very closed to our. The research propose a set three of patterns that introduces variability concept in multi-tenant Software as a Service solutions. The proposed pattern are used to customize the application depending on the tenants. The patterns aim to customize data views, create dynamic menus and implements custom module before or after a data updating.

3 MULTI-TENANCY DATABASE

With cloud advent, NoSQL databases have acquired an increased importance as they are more agile than classical relational database (Sakr et al., 2011) (Chen and Zhang, 2014). The use of NoSQL database is not always the best choice, above all if the application already exists. In this case, the entire database migration would be very long and expensive. Considering that the focus of this work is to adapt legacy applications to cloud, various approaches will be presented to managing multi-tenancy database elaborated by (Chong et al., 2006) and used in several works to better define the concept of multi-tenancy database level. The description of these approaches is proposed through the use of key attributes for a well-designed Saas application (scalability, multi-tenant efficiency and configuration) as presented in (Chong and Carraro, 2006). The approaches proposed are:

- Separated DataBase;
- Shared DataBase, separated schema;
- Shared DataBase, shared schema.

3.1 Separated Database

In this first approach, each tenant has its own database and its set of data that remains logically isolated from data of all the other tenants (See Fig. 1). In this case, in the provisioning process new standard database is created for the new tenant. Each database can be accessed through meta-data services. The database can be modified by the tenant in order to satisfy its needs. The possible modifications are related both to the user interface and the program logic and the tenant can potentially create new fields, new queries, new tables and relationships.

The advantage of this approach relies on the fact that the database can be modified by the tenant so it offers the maximum freedom of extension. Moreover, in some specific case such as sensitive data management (i.e. medical or banking data management) this approach is appropriate due to the strong data isolation requirements. The disadvantages of this approach is related to the fact that it leads to support only a limited number of database for each server and consequence the cost of the infrastructure will be higher.

3.2 Shared Database, Separate Schema

In the second approach a single database is shared by all tenants (see Fig.2). In this database a pre-set of customs fields that tenants can assign and use are

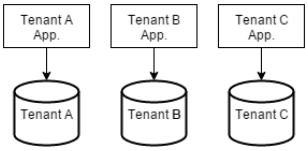

Figure 1: Architecture of ”Separated Database” approach.

present in addition to the standard set fo fields. Each customer can choose what to use these fields for, and how data will be collected for them.

The custom fields can be typed or untyped. The first one enable the customer to use any available built-in type checking and verification functions that the application and database provide to validate the data. In the case of untyped field, the customer can use them to store any type of data (the customer can optionally provide its own validation logic, to prevent users from accidentally entering invalid data).

Shared database allows a single database engine to support a larger number of customers before partitioning becomes necessary rather than isolated approach. This leads to a lower cost of providing service. The disadvantage related to this approach refers to the extensibility of the data model that is limited to the number of provided custom field.

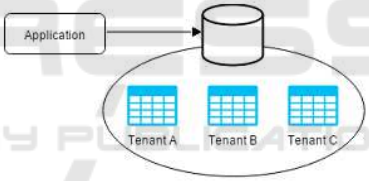

Figure 2: Architecture of ”Shared Database, Separate Schema” approach.

3.3 Shared Database, Shared Schema

A single shared database is building in the third approach (see Fig. 3). In this case customers can extend the data model arbitrarily, storing custom data as namevalue pairs in a separate table. A unique record-ID is assigned to each customer record (including custom data). The unique ID matches one or more rows in a separate extension table. For each row in this table, a name-value pair is stored. The name-value pairs can be created without any limitation (in number and type) by customers.

When the application retrieves a customer record, it performs a lookup in the custom data table, selects all rows corresponding to the customer ID, and returns them to be treated as ordinary data field. In the custom data table, data cannot be typed, because the field may contain data in many different forms for different customers. To solve this, a third column can

optionally hold a data type identifier, so that the data can be cast to the appropriate data type once it is retrieved.

This approach makes the data model arbitrarily extensible, while retaining the cost benefits of using a shared database. But, the added level of complexity for database functions (such as searching, indexing, querying, and updating records) can be considered one disadvantage of this approach. If customers requires considerable degree of flexibility in extending the default data model without the requirement of data isolation. This will be the best approach.

It is important to consider that every time an extensively approach for data model is developed, any extension implemented by a customer will require a corresponding extension to the business logic (so that the application can use the custom data), as well as an extension to the presentation logic (so that users have a way to enter the custom data as input and receive it as output).

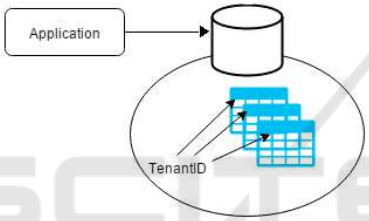

Figure 3: Architecture of ”Shared Database, shared Schema” approach.

4 ENABLE MULTI-TENANCY IN LEGACY APPLICATION

In this section we propose the pattern that we implemented. The methodology used to describe the pattern is described in Figure 4. In the context of OCP project, various organizations have been facing the problem of migration starting from an assessment to evaluate the compatibility of an application with the cloud platform (F. Corradini et al., 2015).

Some problems emerged during the evaluation of this assessment. Among the most common problems that the various applications had to face, it emerged the management of multi-tenancy database that legacy (and single tenant) applications were not able to handle. For this reason, it was decided to develop a pattern that would help various developers in migrating their applications. Pattern validation was done through the use of questionnaires which were then proposed to the same developer and to other industry experts. Depending on the results of the questionnaire, the necessary changes to the pattern were carried out until the questionnaires proposed a satisfactory result. In this case we needed two interaction to validate the pattern. In the first interaction, we focusing our attention on the feedback while the second one was used to refine the pattern.

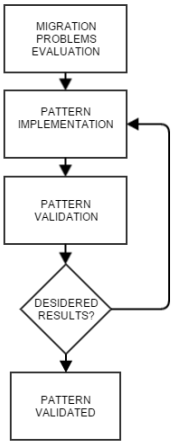

Figure 4: Methodology of pattern definition.

4.1 Pattern

In the previous section, we presented various approaches that literature offers at database management level. In order to enable multi-tenancy in legacy application, database adaptation needs several changes. These changes impact the behaviour of the higher levels. In legacy application the logic layer receives standard data depending on the structure of the tenant. Indeed, multi-tenancy applications data can vary for each tenant as explained before. In this section, we will present a pattern to be applied in an existing legacy application to enable the logic layer to manipulate data from multi-tenancy database. The concept of pattern was adopted in software engineering from the book of the ”Gang of four” (Gamma et al., 1994). This concept relates to provide a solution to a recurring problem with a generic scheme applicable in all contexts. The following information will be used for pattern description:

- The name of the pattern;
- The context of the problem;
- The description of the problem;
- The solution of the problem introduced by:
  1. A class diagram expressed in UML (Object Management Group, 2015a);
  2. An explanation of the solution.
- The consequence of the application of the pattern.

4.1.1 Name

Customizable application for data handling.

4.1.2 Context

The concept of multi tenancy introduced with the advent of cloud has changed the implementation design of applications. With this pattern, we would adapt legacy applications to make them able to manage a multi-tenancy database composed of heterogeneous data among different tenants.

4.1.3 Problem

The multi-tenancy has led to new patterns of database design. Although it is now possible to use a single instance to multiple tenants, hardly the standard solution is suitable for all of them. This involves the management of different information within the same instance. This problem affects both the database and application level. At database level, where it is possible to manage multiple heterogeneous data sources in a single instance, the application must be able to handle different results (different data type or number of arguments) according to the tenant making the query to the database. So, the purpose of this pattern is to make the software adaptable to the tenant needs giving the possibility to change / add / remove fields in standard components of the application.

4.1.4 Solution

This pattern has to be applied to each component of the application that has to be adapted to enable it to manage heterogeneous information. The Component class is the generic class that already exists in the application, with all its attributes and methods already defined. The rest are new classes that do not modify the existing code. *DataConfigurationManagement* class is the class of access to this pattern and it has to manage the properties present in the *Component* class. This class is able to access the generic class through the use of "*addProperty*", "*deleteProperty*" and "*modifyProperty*" functions and it enables to modify the various parameters within the class.

Three methods corresponding to the methods of the class *DataConfigurationManagement* are added to the existing class. The management of the property is done by the class *DynamicProperty*. This class is used to manage all changes to the component and it is used to define whether the property has been changed, added or deleted from the component. The class *DynamicPropertyValue* saves the value of the data and make it accessible to the component. The class *DataComponent* regards the interface of the component with the layer database.

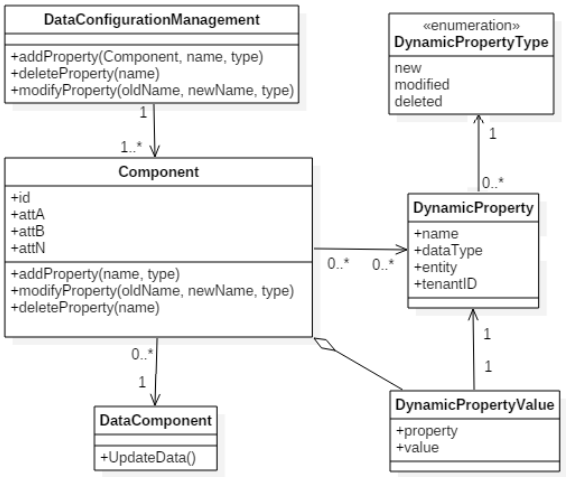

Figure 5: Class Diagram of the pattern that helps to manage different configuration of multi-tenancy db.

4.1.5 Consequences

As mentioned in the previous sections, the impact of the patterns in the existing architecture is low. The pattern integrates itself with the existing code without being invasive and this minimizes changes in what is already done. The *Component* class is the only existing class and it does not require any changes with the exception of the three methods that have to be added. Depending on the implementation of the class *Component* it is possible to implement the pattern using a derived class or to add these methods directly in the class without changing the existing code. In this way, the pattern appears to be context-independent and this makes it applicable every time it is needed.

In addition to the deployment aspect, the pattern has non-functional advantages. First of all, the pattern can increase the deployment process of the developer and it can guarantee the correctness of the system. Another advantage of the pattern is to standardize the applications. In this way a cloud provider can increase the Quality of Services of their hosted application implementing a service that properly manage the pattern.

4.2 Pattern Utilization Example

In this example, we present the utilization of the pattern in order to explain better how it works. First of all, we present the process to add a new property. The steps are showed in Fig. 6 and it can be summarize in these steps:

1. The administrator of the system sends a request of adding property to *DataConfigurationManagement* class;
2. The *DataConfigurationManagement* Class calls the component that has to be modified;
3. The *Component* creates a *DynamicProperty* to handle the request of new property using the method *addProperty*;
4. The *DynamicProporty* class creates *DynamicPropertyValue* class that will be used by *Component* class to logically store data.
5. The *Component* uses the *DataComponent* to store the changes in the DB.

The process of modified and remove standard property is the same as showed for creating new property. The only changes regards the method called in the step 3. To modify an already existing property, it is used the method *modifyProperty*. Indeed, *deleteProperty* is used in order to logically delete property.

The behaviour of the application at execution time does not change. A user, with permissions to access at the component, uses in addition to it all the *DynamicPropertyValue* of its specific tenant (by checking the *tenantID* field of the *DynamicProperty* class). The class *DataComponent* is used to get information from the database or to store modification occurred at application level.

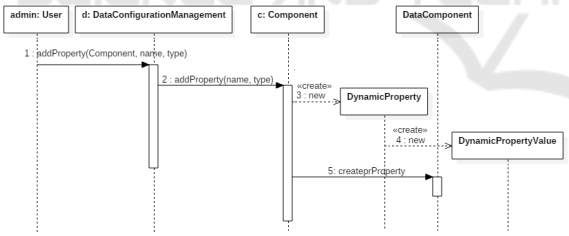

Figure 6: Sequence diagram of new property creation process.

4.3 Validation

The research method used for the validation of the pattern is a survey. The instrument used to collect data is a questionnaire. Questionnaires are considered less time-consuming than interview surveys. The questionnaire was submitted by using Google Form and written in Italian, as all the evaluators are native Italian speakers. The questionnaire consists of a series of closed questions and one open-ended question to provide suggestions to improve the pattern. The questionnaire was proposed to everyone involved in the OCP project with a good technical basis to address the evaluation. The following job functions were selected in order to have a correct evaluation: architect, software development manager or technical manager. A total of 8 experts were involved in the evaluation. The questionnaire consists of 4 different parts:

- Introduction of the questionnaire to explain the context of use of the pattern
- Presentation of the pattern with diagram and description
- Closed-questions
- One open question for criticisms and hints

4.3.1 Questions

In this section we explain the questions used to evaluate the pattern. In the questionnaire eight different characteristics that are needed in a multi-tenancy environment are evaluated. For every characteristic the experts has to assign a score from 1 to 5 depending on the level of satisfiability. The score is expressed in the following terms: **1:** poor; **2:** below average; **3:** average; **4:** good; **5:** excellent. In the Invasiveness and Complexity the score is reverse where score of 1 means excellent and score of 5 means poor. The characteristics taken into account regards both cloud environment and aspects related to the reference application. At the end of the questionnaire there is an open-ended question that is used for some considerations of the experts. The characteristics taken into account in the questionnaire are:

1. **Significance:** Level of importance of the pattern. How useful is to apply it in order to enable multi-tenancy.
2. **Invasiveness:** Degree of impact of the pattern in the existing code. More invasive is the pattern and greater are the difficulties to implements in different contexts.
3. **Effectiveness:** Evaluation of the pattern in relation to the initial troubleshooting. This parameter indicates the degree of solution of the identified problem.
4. **Completeness:** Evaluation of the pattern in relation to the aspects taken into account. Highlight if the pattern covers all possible usage scenarios.
5. **Complexity:** Degree of complexity of implementation of the pattern and degree of efficiency to which the software product can be made available for use.
6. **Portability:** Degree of efficiency for the pattern to be transferred from one environment to another.

7. **Scalability:** Degree of reliability of the pattern when the workload changes. Adaptability of the pattern to work with multiple instances.
8. **Security:** Degree to which the pattern ensures that data are accessible only those authorized to have access.

4.3.2 Results

In this section we describe the results of the last evaluation of the questionnaire(Tab. 1). For each parameter, it is shows the average and the variance of the questionnaire. The average parameter is used to calculate if the result is acceptable, the variance is used to determine if there is an high dispersion of the score in the specific parameter. The final results of the questionnaire are shown as follows:

- **Significance:** The result of this parameter is fundamental because it is used to evaluate if the pattern is useful to enable multi-tenancy in the application. A value of 3.8 indicates that the introduction of this pattern is useful for that aim.
- **Invasiveness:** The average of this parameter is good but the variance is too high. This score indicates a different behaviour depending on the application.
- **Effectiveness:** The result is quite sufficient. The average of the results are quite good but a variance of 0.84 indicates a problem in some type of applications. We will monitor this parameter when we apply the pattern in other application.
- **Completeness:** This results is better than the Effectiveness. Even if the average is the same, in this parameter we have a low level of variance indicating same results from different applications.
- **Complexity:** It is the worst results of the evaluation regarding both average and variance. As in the invasiveness, the results is a consequence of the heterogeneous applications that are taken into account from different experts. The variance in this characteristics indicates a dispersion of the results without a significant majority of a value.
- **Portability:** It is one of the worse results of the evaluation. The average score is sufficient but it has a quite high variance that indicates some ”below average” results.
- **Scalability:** This characteristic is important in cloud and in a multi-tenancy environment. A component can have an elastic workload and it can be duplicated. The scores of this characteristic is good with a low level of variance, this means that the pattern can handle multiple component instances without problem.
- **Security:** It is one of the most important parameter of that pattern. Indeed, it is fundamental that each tenant can access only to their data. The result of this parameter is good with a low variance.

Regarding the first evaluation step, the developers gave some important feedback that help us to improve the pattern. The first critic regards the permission of the instance of *DynamicProperty*. To avoid this problem, we add the attribute *tenantID* who stores the tenant with the permission to access and uses its own specific instance. The second annotation of the developers regards the management of modified or deleted standard properties. In this context, we decided to use an enumerator in *DynamicProperty* class that helps to choose dynamic properties instead of the standard ones during the execution of the component.

Table 1: Results of the pattern evaluation.

Parameter	Average	δ^2
Significance	3.8	0.98
Invasiveness	2.3	1.13
Effectiveness	3.6	0.84
Completeness	3.6	0.27
Complexity	3.1	1.27
Portability	3.5	0.86
Scalability	3.7	0.21
Security	3.9	0.7

5 CONCLUSIONS

In this paper we presented our work about building a pattern to implement in legacy applications in order to enable the accessing multi-tenancy databases. The pattern is proposed as part of the OCP project. It helps developers to modify the code of the application faster and ensure that the applications involved in the project have the same programming logic. However, the pattern is independent of context and therefore can be implemented in any type of application considered that it is independent from the technology. Next step of this work is to extend the pattern to other application involved in OCP in order to have more feedback and refine, if necessary, the pattern. The realization of this pattern was part of a larger effort that leads to the realization of other patterns that solve problems encountered in the valuation of the assessment. The problems are related to the development of applica-

tions that enable a high level of configuration in particular we will focus the attention in these aspects: Role Configuration; Workflow Configuration; Business Rule Configuration; Report Configuration; Action Process Configuration. The final results of this works will be a list of pattern that can be used by the organization involved in OCP project first of all but also all the organization that want migrate legacy application to the cloud.

REFERENCES

Armbrust, M., Fox, O., Griffith, R., Joseph, A. D., Katz, Y., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., et al. (2009). M.: Above the clouds: a berkeley view of cloud computing.

ARTIST Consortium (2015). Artist project. http://www.artist-project.eu/.

Bezemer, C. and Zaidman, A. (2010a). Challenges of reengineering into multi-tenant saas applications. Technical report, Delft University of Technology, Software Engineering Research Group.

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

Bezemer, C.-P., Zaidman, A., Platzbeecker, B., Hurkmans, T., and Hart, A. (2010). Enabling multi-tenancy: An industrial experience report. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–8. IEEE.

Chen, C. P. and Zhang, C.-Y. (2014). Data-intensive applications, challenges, techniques and technologies: A survey on big data. *Information Sciences*, 275:314–347.

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

Chong, F., Carraro, G., and Wolter, R. (2006). Multi-tenant data architecture. *MSDN Library, Microsoft Corporation*, pages 14–30.

Corradini, F., Sabbatini, S., De Angelis, F., Polini, A. (2015). Cloud readiness assessment of legacy application. In *In 5th International Conference on Cloud Computing and Services Science (CLOSER 2015)*.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design patterns: elements of reusable object-oriented software*. Pearson Education.

Jansen, S., Houben, G.-J., and Brinkkemper, S. (2010). *Customization realization in multi-tenant web applications: Case studies from the library sector*. Springer.

Kabbedijk, J., Bezemer, C.-P., Jansen, S., and Zaidman, A. (2015). Defining multi-tenancy: A systematic mapping study on the academic and the industrial perspective. *Journal of Systems and Software*, 100:139–148.

Kabbedijk, J. and Jansen, S. (2011). Variability in multi-tenant environments: architectural design patterns from industry. In *Advances in conceptual modeling. recent developments and new directions*, pages 151–160. Springer.

Kang, S., Kang, S., and Hur, S. (2011). A design of the conceptual architecture for a multitenant saas application platform. In *Computers, Networks, Systems and Industrial Engineering (CNSI), 2011 First ACIS/JNU International Conference on*, pages 462–467. IEEE.

Kwok, T., Nguyen, T., and Lam, L. (2008). A software as a service with multi-tenancy support for an electronic contract management application. In *Services Computing, 2008. SCC'08. IEEE International Conference on*, volume 2, pages 179–186. IEEE.

Mell, P. and Grance, T. (2011). The nist definition of cloud computing.

Mietzner, R., Unger, T., Titze, R., and Leymann, F. (2009). Combining different multi-tenancy patterns in service-oriented applications. In *Enterprise Distributed Object Computing Conference, 2009. EDOC'09. IEEE International*, pages 131–140. IEEE.

Object Management Group (2015a). UML, http://www.uml.org/.

Object Management Group (2015b). Model Driven Architecture. http://www.omg.org/mda/.

OCP Consortium (2015). Open city platform project. http://www.opencityplatform.eu/.

Parkhill, D. F. (1966). Challenge of the computer utility.

REMICS Consortium (2015). Remics project. http://www.remics.eu/.

Sakr, S., Liu, A., Batista, D. M., and Alomari, M. (2011). A survey of large scale data management approaches in cloud environments. *Communications Surveys & Tutorials, IEEE*, 13(3):311–336.

Vaquero, L. M., Rodero-Merino, L., Caceres, J., and Lindner, M. (2008). A break in the clouds: towards a cloud definition. *ACM SIGCOMM Computer Communication Review*, 39(1):50–55.

Wang, Z. H., Guo, C. J., Gao, B., Sun, W., Zhang, Z., and An, W. H. (2008). A study and performance evaluation of the multi-tenant data tier design patterns for service oriented computing. In *e-Business Engineering, 2008. ICEBE'08. IEEE International Conference on*, pages 94–101. IEEE.