Delft University of Technology
Software Engineering Research Group
Technical Report Series

# Enabling Multi-Tenancy: An Industrial Experience Report

Cor-Paul Bezemer, Andy Zaidman, Bart Platzbeecker, Toine Hurkmans and Aad 't Hart

Report TUD-SERG-2010-030

TU Delft

SERG

TUD-SERG-2010-030

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
http://www.se.ewi.tudelft.nl/techreports/

For more information about the Software Engineering Research Group:
http://www.se.ewi.tudelft.nl/

Note: Accepted for publication in the Proceedings of the 26th IEEE Int. Conf. on Software Maintenance (ICSM), 2010, IEEE.

# Enabling Multi-Tenancy: An Industrial Experience Report

Cor-Paul Bezemer and Andy Zaidman
Faculty of EEMCS
Delft University of Technology, The Netherlands
Email: {c.bezemer, a.e.zaidman}@tudelft.nl

Bart Platzbeecker, Toine Hurkmans and Aad 't Hart
Research & Innovation
Exact, The Netherlands
Email: {bart.platzbeecker, toine.hurkmans, aad.hart}@exact.com

***Abstract*—Multi-tenancy is a relatively new software architecture principle in the realm of the Software as a Service (SaaS) business model. It allows to make full use of the economy of scale, as multiple customers – "tenants" – share the same application and database instance. All the while, the tenants enjoy a highly configurable application, making it appear that the application is deployed on a dedicated server. The major benefits of multi-tenancy are increased utilization of hardware resources and improved ease of maintenance, resulting in lower overall application costs, making the technology attractive for service providers targeting small and medium enterprises (SME). Therefore, migrating existing single-tenant to multi-tenant applications can be interesting for SaaS software companies. In this paper we report on our experiences with reengineering an existing industrial, single-tenant software system into a multi-tenant one using a lightweight reengineering approach.**

## I. Introduction

Software as a Service (SaaS) represents a novel paradigm and business model expressing the fact that companies do not have to purchase and maintain their own ICT infrastructure, but instead, acquire the services embodied by software from a third party [1], [2]. The customers subscribe to the software and underlying ICT infrastructure (service on-demand) and require only Internet access to use the services. The service provider offers the software service and maintains the application [3]. However, in order for the service provider to make full use of the economy of scale, the service should be hosted following a multi-tenant model [4].

Multi-tenancy is an architectural pattern in which a single instance of the software is run on the service provider's infrastructure, and multiple tenants access the same instance. In contrast to the multi-user model, multi-tenancy requires customizing the single instance according to the multi-faceted requirements of many tenants [4]. The multi-tenant model also contrasts the multi-instance model, in which each tenant gets his own (virtualized) instance of the application [5]. We define a multi-tenant application and tenant as the following [6], [8]:

**Definition 1.** *A **multi-tenant** application lets customers (**tenants**) share the same hardware resources, by offering them one shared application and database instance, while allowing them to configure the application to fit their needs as if it runs on a dedicated environment.*

**Definition 2.** *A **tenant** is the organizational entity which rents a multi-tenant SaaS solution. Typically, a tenant groups a number of users, which are the stakeholders in the organization.*

The benefits of the multi-tenant model are twofold. On one hand, application deployment becomes easier for the service provider, as only one application instance has to be deployed, instead of hundreds or thousands. On the other hand, the utilization rate of the hardware can be improved as multiple tenants share the same hardware resources[1]. These two factors make it possible to reduce the overall costs of the application and this makes multi-tenant applications especially interesting for customers in the small and medium enterprise (SME) segment of the market, as they often have limited financial resources and do not need the computational power of a dedicated server.

Because of these benefits, many organizations working with SaaS technology are currently looking into transforming their single-tenant applications into multi-tenant ones. Yet, they see a major barrier in the reengineering process that they should go through for this transformation [7]. In previous work, we have proposed a lightweight approach for carrying out this process in a structured manner and applied it to a small-scale open-source software project [8]. In this paper, we present a case study of applying this approach to an industrial application. Our main aims are to show that migrating from a single-tenant setup to a multi-tenant one can be done (1) easily, in a cost-effective way, (2) transparently for the end-user and (3) with little effect for the developer, as the adaptations are confined to small portions of the system, creating no urgent need to retrain all developers.

The structure of this paper is as follows: first, we briefly introduce Exact, our industrial partner where we have performed our case study. In Section III, we provide a summary of our reengineering pattern which supports the transformation of a single-tenant into a multi-tenant application. In Section IV we describe the industrial target application which we migrated using this pattern. The actual case study is dealt with in Section V. We then discuss our findings and their threats to validity in Section VI, before detailing related work in Section VII. Section VIII presents our conclusions and ideas for future work.

[1]Please note that virtualization would also enable improved hardware utilization, but would not solve issues with maintenance, as each virtualized instance should be maintained.

## II. Exact and Multi-Tenancy

Exact[2] is a Dutch-based software company, which specializes in enterprise resource planning (ERP), customer relationship management (CRM) and financial administration software. Exact has over 2200 employees working in more than 40 countries. Founded in 1984, Exact has over 25 years of experience in multi-user client/server software and web applications. Since several years, Exact has also been offering a successful multi-tenant SaaS solution.

Multi-tenancy is an attractive concept for Exact because they target the SME segment of the market. By having the opportunity to share resources between customers, services can be offered to the customers at a lower overall price. In addition, maintenance becomes easier — and thus cheaper — as less different instances must be maintained.

One of the research projects of the Research and Innovation department of Exact is to migrate an existing single-tenant to a multi-tenant application. In the next section, we explain the reengineering pattern that we used for this migration, while Sections IV and V deal with respectively the target system, Codename, and the actual case study.

## III. Multi-Tenancy Reengineering Pattern

Our multi-tenancy reengineering pattern [8] takes into account some of the key aspects of multi-tenancy:

1) The possibility to share hardware resources, enabling cost reductions [9], [10].
2) A high degree of configurability, enabling each customer to create his own look-and-feel and workflow within the application [11], [12], [13].
3) A shared application and database instance, enabling easier maintenance [4].

Figure 1 provides an overview of the multi-tenancy reengineering pattern that we have previously developed [8]. The primary goals of the pattern are the following:

1) Migrate a single-tenant to a multi-tenant application with *minor* adjustments in the existing business logic.
2) Let application developers be unaware of the fact that the application is multi-tenant.
3) Clearly separate multi-tenant components, so that monitoring and load balancing mechanisms can be integrated in the future.

In order to reach our goals, our reengineering pattern requires the insertion of three components in the target application. The remainder of this section will explain the importance and the requirements of each of these components.

### A. Authentication

*Motivation.* Because a multi-tenant application has only one application and database instance, all tenants use the same physical environment. In order to be able to offer customization of this environment and to make sure that tenants can only access their own data, tenants must be authenticated. While user authentication is possibly already present in the target application, a separate tenant-specific authentication mechanism might be required, for two reasons: (1) it is usually much easier to introduce an additional authentication mechanism, then to change the existing one, and (2) tenant authentication allows a single user to be part of more than one logical organization, which extends the idea of user authentication with "groups". A typical example of such a situation would be a bookkeeper, who works for multiple organizations.

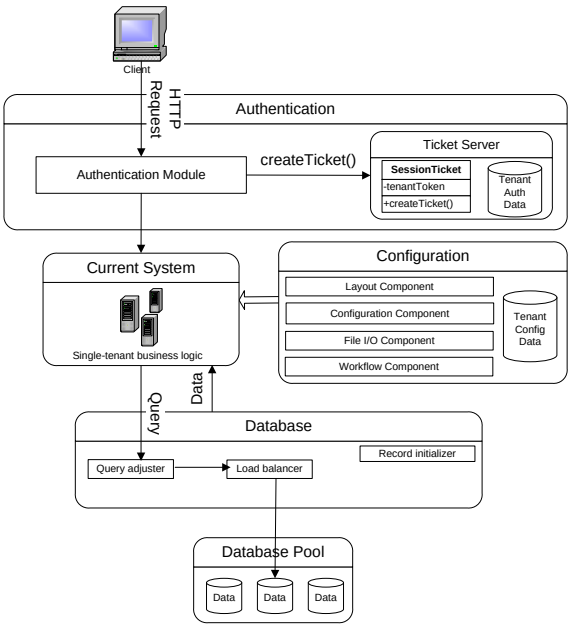


Fig. 1. Multi-tenancy reengineering pattern

*Implementation.* The authentication component provides the mechanism required to identify a tenant throughout the application, by generating a session ticket after a tenant successfully logs in. The correct application configuration is loaded based on the values in this ticket. Note that this mechanism does not interfere with the authentication logic of the single-tenant application, which means that any security measures implemented in this logic are still in order.

### B. Configuration

*Motivation.* In a single-tenant environment, every tenant has his own, (possibly) customized application instance. In multi-tenancy, all tenants share the same application instance, although it must appear to them as if they are using a dedicated one. Because of this, a key requirement of multi-tenant applications is the possibility to configure and/or customize the application to a tenant's need [12].

In single-tenant software, customization is often done by creating branches in the development tree. In multi-tenancy this is no longer possible and customization must be made possible through configuration [11].

[2]http://www.exact.com

*Implementation.* In order to enable multi-tenancy and let the user have a user-experience as if he were working in a dedicated environment, it is necessary to allow at least the following types of configuration:

*1) Layout Style:* Layout style configuration allows the use of tenant-specific themes and styles.

*2) General Configuration:* The general configuration component allows the specification of tenant-specific configuration, encryption key settings and personal profile details.

*3) File I/O:* The file I/O configuration component allows the specification of tenant-specific file paths, which can be used for, e.g., report generation.

*4) Workflow:* The workflow configuration component allows the configuration of tenant-specific workflows. An example of an application in which workflow configuration is required is an ERP application, in which the workflow of requests can vary significantly for different tenants.

### C. Database

*Motivation.* Because all tenants use the same database instance, it is necessary to make sure that they can only access their own data.

*Implementation.* Because current off-the-shelf DBMSs are not capable of dealing with multi-tenancy themselves [14], this should be done in a layer between the business logic and the application's database pool. The main tasks of this layer are as follows:

*1) Creation of new tenants in the database:* If the application stores and/or retrieves data, which can be made tenant-specific, in/from a database, it is the task of the database layer to create the corresponding database records when a new tenant has signed up for the application.

*2) Query adaptation:* In order to provide adequate data isolation, the database layer must make sure that all queries are adjusted so that each tenant can only access his own records.

*3) Load balancing:* To improve the performance of the multi-tenant application, efficient load balancing is required for the database pool. Any Service Level Agreements (SLAs) [15] or financial data legislation should be taken into account.

## IV. Exact Codename

As stated in Section II, multi-tenancy is an attractive concept for Exact, because it allows Exact to deliver highly customized applications to their SME customers at an attractive price, which can be reached through the economy of scale benefits that multi-tenancy offers. While Exact has experience with multi-tenancy, they also have existing single-tenant applications that they want to transform into multi-tenant ones.

One of these applications is a research prototype, dubbed Exact Codename. Codename is a proof of concept, single-tenant widget framework that offers the possibility of creating software solutions using widgets as building blocks. The Exact research team has been working for 4 years on Codename and it is the intention to integrate parts of Codename in commercial Exact products in the short to medium term future.

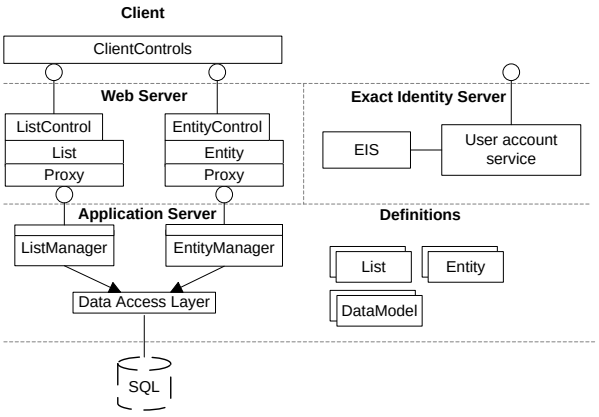


Fig. 2. Architecture of Exact Codename

Codename is being developed in C# and ASP.NET and consists of approximately 165K lines of code. Figure 2 depicts the (simplified) architecture of Codename.

### A. Architecture of Codename

Codename is built upon two major concepts, the *List* and the *Entity*. A list represents a list of data, such as a list of documents. An entity represents an object, such as News (a news item).

An entity and a list are described using a domain specific language and the descriptions are currently stored in definition files. These definitions are stored separately from the framework code, which allows them to be edited by non-technical domain experts. Such a definition file may contain details about how to retrieve the entity or list from the database, or behavior. For example, the definition of News contains details on how a News item can be found in the database, and it also tells us that News is a type of Document (which is itself an entity). The default HTML layout of an entity or list is also stored in a (separate) definition file.

Because an entity or list can be created using a definition file only, it is easy for domain experts to add new or edit existing entities or lists.

On the application server, the *ListManager* and *EntityManager* can be used to instantiate a new list or entity. When a new list or entity is created, these managers read the corresponding definition file and generate the required object. All database access is done through the *Data Access Layer*. To allow the use of multiple data sources, possibly in different formats, logical names are used for database columns or tables rather than the physical names. In the Data Access Layer, these logical names are translated to physical names (using the *DataModel* definitions).

The web server communicates with the application server using Windows Communication Foundation (WCF) services and a proxy. The goal of the web server is to generate HTML and JavaScript (JS) representations of the lists and entities for the client. A client can request a list or entity using the *ListControl* or *EntityControl* web services. The client can only

retrieve data from or write data to the database using these two services.

### B. Exact Identity Server

A separate component in Codename's architecture is the Exact Identity Server (EIS), which is an implementation of the Microsoft Identity Foundation. In the EIS a token is generated when a tenant successfully logs in to the system. This (encrypted) token contains enough information to identify the tenant throughout the system without contacting the EIS again. This allows single sign-on (SSO) for multiple Exact applications (*relying parties*). The protocol used to do this is SAML 1.1. A token contains several claims, such as the Globally Unique Identifier (GUID) of the user which is logged in. The EIS offers a User Account Service as well, which allows relying parties to add their own users to the EIS.

## V. Case Study: Codename$^{MT}$

In this section, we present our case study of enabling multi-tenancy in a single-tenant application using the multi-tenancy reengineering pattern that we discussed in Section III. Our target application is Codename, of which we gave an overview in Section IV.

### A. Motivation

In addition to the general advantages of multi-tenancy [6], [8], being able to reengineer existing single-tenant applications into multi-tenant ones is interesting for a number of reasons:

1) Existing business logic can be reused with minor adaptations.
2) As our reengineering pattern is lightweight and requires minor adaptations only, most developers will not be aware of the fact that the application is multi-tenant, which means that not all developers need to be trained in multi-tenancy.
3) Lessons learned from applying a pattern may lead to improvements in the architecture of existing multi-tenant products.

### B. Applying the Multi-Tenancy Pattern

In our case study, we will apply our multi-tenancy reengineering pattern to Codename, resulting in a multi-tenant application Codename$^{MT}$. For transforming Codename into Codename$^{MT}$, we are introducing the components that we have explained in Section III into Codename.

*1) Authentication:* As identifying to which tenant a user belongs can be done using the tenant's ID only, the existing authentication mechanism could easily be extended. We added Codename$^{MT}$ to the EIS as a relying party, so that we could add users for this application to EIS. After this, we extended the Codename `User` object with a `TenantID` property, which is read from the token after a user successfully logs in. Because the User object is globally available throughout Codename, the `TenantID` is available globally as well. Note that EIS does not keep track of tenant details other than the `TenantID`. Currently this is the task of the relying party.

```
// attach event
protected void Application_PreRequestHandlerExecute(
 object s, EventArgs e){
   Page p = this.Context.Handler as Page;
   p.PreInit += new EventHandler(page_PreInit);
}

// set tenant-specific theme and master page
protected void page_PreInit(object s, EventArgs e){
   Page p = this.Context.Handler as Page;
   p.Theme = TenantContext.GetTenantTheme();
   p.MasterPageFile = TenantContext.GetTenantMasterpage();
}
```

Fig. 3. Dynamically setting the tenant-specific style

After our adaptations, an EIS token for the Codename$^{MT}$ application contains a `GUID` and a `TenantID`. The `TenantID` is used to identify the tenant to which the owner of the token belongs. The `GUID` is used to identify the user within Codename$^{MT}$. Note that the user identification process is unchanged compared to the process in Codename, leaving any values like security levels intact.

*2) Configuration:* While applying the pattern to the single-tenant configuration, we limited our case study to the degree of configuration currently possible in Codename. In contrast to the pattern, Codename$^{MT}$ stores all configuration data in the application database, rather than in a separate database.

*a) Layout Style:* In Codename, the layout style of the application is managed by the following:

- ASP.NET master pages
- ASP.NET themes

The .NET framework offers the possibility to dynamically change these by attaching an event early in the page lifecycle. We have adapted the global.asax[3] file of the application with the code depicted in Figure 3, which loads the tenant-specific style for each page request.

*b) General Configuration:* All general configuration, e.g. profile settings, in Codename is stored in the database. This means that making the configuration tables multi-tenant also makes the general configuration multi-tenant.

*c) File I/O:* The only file I/O used in Codename is to load the definition files on the application server. Originally these definition files were loaded from the `xmd/list` and `xmd/entity` directories. We have adapted this code to check if the `xmd/tenantID/list` or `xmd/tenantID/entity` directory contains the requested file. If it exists, the tenant-specific file is loaded, otherwise, a default file is loaded. We have implemented this mechanism to allow tenants to decide whether they want to configure their own lists and entities or use the defaults. Codename also implements a caching system for definition files, which we have adapted to be aware of the existence of tenant-specific definitions.

*d) Workflow:* In Codename, the application workflow can currently only be configured by physically changing the .aspx page, which describes the process, so that it uses the

[3]In ASP.NET, the (optional) global.asax file is used to access session and application-level events.

| Type of query | Query extension |
|---|---|
| SELECT | Add Filter('TenantID', 123) |
| JOIN | Add Filter('TenantID', 123) |
| UPDATE | Add Filter('TenantID', 123) |
| DELETE | Add Filter('TenantID', 123) |
| INSERT | Add Field('TenantID', 123) |

TABLE I
MULTI-TENANT QUERY EXTENSIONS FOR TENANTID '123'

required library. While tenant-specific workflow configuration using this approach was included in the case study design, the implementation remains future work.

Codename uses a URL rewriting mechanism to allow application users to request URLs which contain less machine code (*friendly URLs*). This leads to better readable URLs such as `docs/person/corpaul` instead of `?page=person&id={12345-abcde-890}`. By altering this rewriting module to load a tenant-specific .aspx page, workflow configuration can be established.

*3) Database:* All database queries in Codename are generated using the Data Access Layer, so that metadata stored in the data model definitions can always be used during query generation. Because all queries are created in one component, automatically extending them to use the `TenantID` is straightforward. To prevent unnecessary duplication of data, we added the property `IsMultiTenant` to the data model. Setting this property to false indicates that data in the table is not tenant-specific, such as country ISO codes or postal shipping rates. This allows us to generate more efficient queries. We added a `TenantID` column to the tables that were specified as multi-tenant.

After this, we adapted the module which generates the query. For each queried table, the table metadata is retrieved from the data model to see whether the table contains tenant-specific data. If this is the case, the query is extended using the extensions depicted in Table I. Note that for all subqueries and each `JOIN` clause in a `SELECT` query, the same occurs. In the Data Access Layer, a `Filter` adds a criterion to the `WHERE` clause of a query and a `Field` adds a column update to the `SET` clause of a query.

Future work regarding the database component includes adding usage of the `TenantID` to indexes on tables that contain multi-tenant data.

In this case study, we did not implement automatic creation of new tenants in the database. We plan on doing this when the signup process is linked with the EIS User Account Service. In addition, we did not implement load balancing. This is a very difficult task due to the number and complexity of constraints in financial software, e.g., because of the legislation of several countries on where financial data may be stored. Load balancing in a multi-tenant application will be addressed in future research.

### C. Evaluation

For testing whether our reengineering pattern that transformed Codename into Codename$^{MT}$ did not break any of the major functionalities in Codename, we followed a double approach using code reviews and manual tests. As such, we performed a code review together with the third author of this paper, one of the lead architects of the Exact research team. Furthermore, we manually tested the most important functionality of the application. While we consider manual testing to be sufficient for this particular case study, amongst others due to the support from Exact, we do acknowledge that automated testing is a necessity, which is why we aim to investigate an automated test methodology for multi-tenant applications in future research.

For the actual testing of Codename$^{MT}$ we first added two test users with different `TenantIDs` on the EIS. Then we created tenant-specific themes and master pages and verified that they were loaded correctly after logging the test users in. After this, we created a number of tenant-specific definition files and verified that the correct ones (including default files) were loaded.

To test the database component, we have assigned different documents to each test user and verified the correct ones were shown in document listings after logging in. In addition, we have verified that queries were extended correctly by manually inspecting a random subset of queries taken from a SQL Server Profiler trace, recorded during usage of the application.

Our double approach where we combined code reviews and manual tests to verify whether Codename$^{MT}$ did not break any of the major functionality from Codename yielded no reports of any faults.

## VI. LESSONS LEARNED & DISCUSSION

In this paper we have applied our reengineering pattern that guides the reengineering of single-tenant applications into multi-tenant ones. In previous work [8], we have applied this pattern on a small-scale open source wiki system, and in this paper we report on our experiences with the reengineering pattern in an industrial environment. We will now touch upon some of the key lessons that we have learned when applying our reengineering pattern.

### A. Lessons learned

*a) Lightweight reengineering approach:* We have applied our multi-tenancy reengineering pattern by extending the original Codename code with approximately 100 lines of code, thus transforming it into Codename$^{MT}$. This shows that our pattern can assist in carrying out the reengineering process in an efficient way, with relatively little effort. In our case study, the reengineering could be done in five days, without prior knowledge of the application, but with the help of domain experts from Exact. The ease by which we were able to reengineer the original Codename into Codename$^{MT}$ is of interest to our industrial partner Exact, and other companies alike, as it shows that even the initial costs of migrating towards multi-tenancy are relatively low and should thus not be seen as a barrier.

*b) Importance of architecture:* While not surprising, another lesson we learned from the migration was that having a layered architecture is essential, both for keeping our reengineering approach lightweight and for doing the reengineering quickly and efficiently [16]. Without a well-layered architecture, applying our pattern would have taken much more effort.

*c) Automated reengineering proves difficult:* The ease by which we were able to reengineer Codename automatically raises the question whether it is possible to automate the reengineering process. Unfortunately, we think this is very difficult to achieve, as the reengineering requires a considerable amount of architectural and domain knowledge of the application, which is difficult and costly to capture in a reengineering tool. Furthermore, the integration of the components of our multi-tenancy pattern is strongly dependent on the implementation of the existing application. A similar observation about the difficulty to automate design pattern detection and reengineering approaches was made by Guéhéneuc and Albin-Amiot in [17]. Specifically in our industrial environment, the architectural and domain knowledge of the lead architect of Codename — the third author of this paper —, proved very valuable for the quick and efficient reengineering of the target application. Capturing this tacit knowledge in an automatic reengineering tool would prove difficult and expensive.

*d) Fully transparent for the end-user:* An interesting observation is that no changes had to be made to the client side of the application, i.e., in terms of JavaScript. This serves a first indication that the end-user will not be aware of the fact that he is using a multi-tenant application instead of a single-tenant one. Furthermore, the (manual) tests have also shown that the other parts of the user interface have not evolved when going from Codename to Codename$^{MT}$.

*e) Little effect for the developer:* Because we could enable multi-tenancy by making small changes only, most developers can remain relatively uneducated on the technical details. For example, they do not have to take multi-tenancy into account while writing new database queries as these are adapted automatically.

### B. Discussion

In this version of Codename$^{MT}$ we did not implement workflow configuration. The reason for this is that we limited our case study to the degree of configuration currently possible in Codename. A first step towards workflow configuration is to implement the tenant-specific friendly URL mechanism as described in Section V-B2d. This approach still requires the tenant (or an Exact developer) to develop a custom .aspx page. In a future version of Codename$^{MT}$, Exact is aiming at making workflow configuration possible by enabling and disabling modules and widgets using a web-based administration, rather than requiring a tenant to make changes to an .aspx page.

We have applied our pattern by modifying existing single-tenant code. One may argue that multi-tenant code additions should be completely isolated, e.g., by integrating the code using aspect-oriented programming. As typical aspect-oriented programming (following the AspectJ model) does not offer a fine enough pointcut mechanism to target all join points that we would need to change, we decided not to use aspects. Please note however, that using aspect-oriented programming would become applicable after a thorough refactoring of the source code, but this was beyond the scope of the lightweight reengineering pattern that we intended for.

### C. Threats to Validity

We were able to apply our multi-tenancy pattern with relatively little effort. One of the reasons for this is the well-designed and layered architecture of Codename. In our previous case study on an open source wiki system called ScrewTurn [8], applying the pattern took more time because of the lack of architectural documentation and knowledge. In addition, the existing integration of the authentication using EIS and the possibility to add a TenantID claim to the token considerably shortened the implementation time for the authentication component. Finally, the database component could be adapted relatively easily as well, as all queries are created in one single component, i.e., the Data Access Layer, which made searching for query generations throughout the application superfluous. As such, we acknowledge that Codename might not be representative for all systems, but we also acknowledge that having intimate knowledge of the application is equally important for the reengineering of single-tenant into multi-tenant applications. Another confounding factor is the complexity of both the source code and the database schema, as both can have a direct influence on the ease by which an existing single-tenant application can be reengineered.

We have manually verified the correctness of the implementation of the functionality in Codename$^{MT}$. While we are confident that the verification was done thoroughly and was supported by one of the lead architects of Codename, we do see the need for automatic testing in this context. As such, we consider investigating the possibilities of defining a test methodology for multi-tenant applications as future work.

The case study we have conducted is not complete yet. For example, we have not implemented workflow configuration and automated tenant creation. As it is possible that these implementations introduce performance penalties, we did not formally evaluate the performance overhead of our approach. Although we have not encountered performance drawbacks yet, we consider a formal evaluation of the performance as future work.

### D. Multi-Tenancy in the Real World

Although the benefits of multi-tenancy are obvious, there are some challenges which should be considered before implementing it.

Because all tenants use the same hardware resources, a problem caused by one tenant affects all the others. Additionally, the data of all tenants is on the same server. This results in a more urgent requirement for scalability, security and zero-downtime measures than in single-tenant software.

Finally, because multi-tenancy requires a higher degree of configurability, the code inherently becomes more complex,

which may result in more difficult software maintenance if not implemented correctly [6].

## VII. Related work

Most research in the field of reengineering in the area of "service-oriented software systems" has focused on approaches to migrate, port and wrap legacy assets to web services. Two notable examples in this context are the works of Sneed and Canfora et al. Sneed reports on an approach to wrap legacy code behind an XML shell [18]. Sneed's approach allows individual legacy functions to be offered as web services to any external user. The approach has been applied successfully to the integration of both COBOL and C++ programs in a service-oriented system. Canfora et al. presented an approach to migrate form-based software systems to a service [19]. The approach provides a wrapper that encapsulates the original user interface and interacts with the legacy system which runs within an application server.

We are currently not aware of any research that investigates the reengineering of the first generation of service-oriented systems, an area that we believe to be an important one, as many of the first generation service-based systems have carried over some of the flaws from the systems from which they originate. In particular, we are not aware of any multi-tenancy reengineering strategies.

That being said, multi-tenancy is a relatively new paradigm and a number of papers have been written on different models of multi-tenancy (e.g., the work of Kwok et al. [4]) and the performance of multi-tenancy [9].

## VIII. Conclusion

In this paper, we have applied our lightweight multi-tenancy reengineering pattern to Codename, an industrial single-tenant application engineered by Exact. This reengineering pattern is a guiding process that allows to quickly and efficiently transform a single-tenant application into a multi-tenant one, thereby also providing capabilities for tenant-specific layout styles, configuration and data management. The result is Codename$^{MT}$, a multi-tenant version of Codename, offering the typical benefits of multi-tenancy, i.e., increased usage of hardware resources and easier maintenance.

Our case study has learned us that our approach:

1) Is lightweight, as implementation was done in about 100 lines of code, which took approximately 5 days to implement. This makes the approach attractive for Exact and other companies, because of the low initial investments. On a side note, we do observe that having a nicely layered architecture is a benefit for doing the migration quickly and efficiently.
2) Is transparent to the end-user, as (1) the look-and-feel of the application does not need to be changed and (2) the end-user does not know that the application is multi-tenant.
3) Does not require all developers working on the project to be trained in multi-tenancy, as the changes to the code are minimal and confined to some small parts.

As important directions for future work, we see the development of a test methodology and a real-time monitoring mechanism for multi-tenant applications. The former will enable to determine the optimal moment for online software evolution in the face of zero-downtime for customers, while the latter is essential when tackling larger reengineering efforts in the realm of multi-tenancy. In addition, we will continue to work with Exact on extending the configuration options for Codename$^{MT}$, in particular, the workflow configuration support.

## Acknowledgment

The authors would like to thank Exact for providing the funds and opportunity to perform this research. Further support came from the *NWO Jacquard ScaleItUp* project.

## References

[1] N. Gold, C. Knight, A. Mohan, and M. Munro, "Understanding service-oriented software," *IEEE Software*, vol. 21, no. 2, pp. 71–77, 2004.

[2] M. Turner, D. Budgen, and P. Brereton, "Turning software into a service," *Computer*, vol. 36, no. 10, pp. 38–44, 2003.

[3] J. M. Kaplan, "Saas: Friend or foe?" in *Business Communications Review*, June 2007, pp. 48–53, http://www.webtorials.com/abstracts/BCR125.htm.

[4] T. Kwok, T. Nguyen, and L. Lam, "A software as a service with multi-tenancy support for an electronic contract management application," in *Proceedings of the International Conference on Services Computing (SCC)*. IEEE Computer Society, 2008, pp. 179–186.

[5] F. Chong, G. Carraro, and R. Wolter, "Multi-tenant data architecture," http://msdn.microsoft.com/en-us/library/aa479086.aspx, June 2006.

[6] C.-P. Bezemer and A. Zaidman, "Multi-tenant saas applications: Maintenance dream or nightmare?" in *Proceedings of the 4th International Joint ERCIM/IWPSE Symposium on Software Evolution (IWPSE-EVOL)*. ACM, 2010, p. To appear.

[7] C.-H. Tsai, Y. Ruan, S. Sahu, A. Shaikh, and K. G. Shin, "Virtualization-based techniques for enabling multi-tenant management tools." in *18th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM)*, ser. LNCS, vol. 4785. Springer, 2007, pp. 171–182.

[8] C.-P. Bezemer and A. Zaidman, "Challenges of reengineering into multi-tenant saas applications," Delft University of Technology, Tech. Rep. TUD-SERG-2010-012, 2010.

[9] Z. H. Wang, C. J. Guo, B. Gao, W. Sun, Z. Zhang, and W. H. An, "A study and performance evaluation of the multi-tenant data tier design patterns for service oriented computing," in *Proceedings of the International Conference on e-Business Engineering (ICEBE)*. IEEE Computer Society, 2008, pp. 94–101.

[10] B. Warfield, "Multitenancy can have a 16:1 cost advantage over single-tenant," http://smoothspan.wordpress.com/2007/10/28/multitenancy-can-have-a-161-cost-advantage-over-single-tenant/ (last visited on May 20th, 2010), October 2007.

[11] Nitu, "Configurability in SaaS (software as a service) applications," in *Proceedings of the 2nd annual India Software Engineering Conference (ISEC)*. ACM, 2009, pp. 19–26.

[12] S. Jansen, G.-J. Houben, and S. Brinkkemper, "Customization realization in multi-tenant web applications: Case studies from the library sector," in *Proceedings of the 10th International Conference on Web Engineering (ICWE)*, ser. LNCS, vol. 6189. Springer, 2010, pp. 445–459.

[13] J. Müller, J. Krüger, S. Enderlein, M. Helmich, and A. Zeier, "Customizing enterprise software as a service applications: Back-end extension in a multi-tenancy environment," in *Proceedings of the 11th International Conference on Enterprise Information Systems (ICEIS)*, ser. Lecture Notes in Business Information Processing, vol. 24. Springer, 2009, pp. 66–77.

[14] D. Jacobs and S. Aulbach, "Ruminations on multi-tenant databases," in *Datenbanksysteme in Business, Technologie und Web (BTW), 12. Fachtagung des GI-Fachbereichs Datenbanken und Informationssysteme (DBIS), Proc. 7.-9. Mrz*, ser. LNI, vol. 103. GI, 2007, pp. 514–521.

[15] X. H. Li, T. Liu, Y. Li, and Y. Chen, "Spin: Service performance isolation infrastructure in multi-tenancy environment," in *Proceedings of the 6th International Conference on Service-Oriented Computing (ICSOC)*, ser. LNCS, vol. 5364. Springer, 2008, pp. 649–663.

[16] P. Laine, "The role of SW architecture in solving fundamental problems in object-oriented development of large embedded SW systems," in *Proceedings of the IEEE/IFIP Working Conference on Software Architecture (WICSA)*. IEEE Computer Society, 2001, pp. 14–23.

[17] Y.-G. Guéhéneuc and H. Albin-Amiot, "Using design patterns and constraints to automate the detection and correction of inter-class design defects," in *Proceedings of the International Conference on Technology of Object-Oriented Languages (TOOLS)*. IEEE Computer Society, 2001, pp. 296–306.

[18] H. M. Sneed, "Integrating legacy software into a service oriented architecture," in *Proceedings of the Conference on Software Maintenance and Reengineering*. IEEE Computer Society, 2006, pp. 3–14.

[19] G. Canfora, A. R. Fasolino, G. Frattolillo, and P. Tramontana, "A wrapping approach for migrating legacy system interactive functionalities to service oriented architectures," *Journal of Systems and Software*, vol. 81, no. 4, pp. 463–480, 2008.

TUD-SERG-2010-030
ISSN 1872-5392