CHAPTER

REENGINEERING DATA-CENTRIC INFORMATION SYSTEMS FOR THE CLOUD – A METHOD AND ARCHITECTURAL PATTERNS PROMOTING MULTITENANCY

13

Andrei Furda[*], **Colin Fidge**[*], **Alistair Barros**[*], **Olaf Zimmermann**[†]
[*]*Science and Engineering Faculty, Queensland University of Technology (QUT), Brisbane, QLD, Australia*
[†]*University of Applied Sciences of Eastern Switzerland (HSR FHO), Rapperswil, Switzerland*

13.1 INTRODUCTION

The Software-as-a-Service (SaaS) deployment model in cloud computing has revolutionized the way enterprises use software applications. SaaS and cloud computing have opened paths to new and unique business opportunities for both large enterprises and start-ups [31,38].

SaaS applications reach their full potentials with respect to cost efficiency and on-demand elasticity when they support *multitenancy* at the application level, i.e., *native multitenancy*. This chapter focuses on this native multitenancy at the application level (as opposed to multiple-instance multitenancy at the virtualized infrastructure level). Multitenant SaaS applications enable multiple groups of consumers (e.g., organizations, departments) to simultaneously access and share application instances [11]. A multitenant SaaS deployment often reduces software maintenance and deployment costs compared to multiple-instance applications that would require a separate application instance for each particular tenant [1,6].

Many traditional enterprise applications were developed before the cloud and SaaS era and were intended to be used by a single organization. Therefore, such applications usually do not support multitenancy. Nevertheless, the combination of a typically large financial investment in legacy enterprise applications, on the one hand, and the high benefits expected from the SaaS business model, on the other hand, motivate architects to reengineer existing enterprise applications to support multitenancy and to deploy them as multitenant SaaS applications in cloud environments [8].

In a multitenant application, confidential data belonging to individual tenants is processed and stored by a shared application instance. Therefore, the reengineering process needs to ensure strict separation of confidential data. It is absolutely crucial that multitenant applications do not allow any tenant's data to be read or modified by other, unauthorized tenants that share the same application instance.

This chapter presents a method and a set of architectural patterns for systematically reengineering data-sensitive enterprise applications into secure multitenant software services that can be deployed to public and private cloud offerings seamlessly. The presented method covers both architectural refactorings and code-level considerations, including appropriate testing and code reviewing techniques that focus on typical defects in multitenant applications.

The chapter is organized as follows. Section 13.2 explains the context of multitenancy in cloud computing. Section 13.3 describes the necessary steps in multitenant refactoring from planning to execution to validation (including testing and code reviews). Section 13.4 introduces the architectural refactoring method and presents architectural solution details. Section 13.5 then describes solution details on testing and code review methods. Section 13.6 presents a fictitious, but realistic and representative, case study implementation distilled from real-world requirements and application architectures, Section 13.7 discusses the research results, and Section 13.8 covers related work. Section 13.9 concludes the chapter with a summary and outlook on future work.

13.2 CONTEXT AND PROBLEM: MULTITENANCY IN CLOUD COMPUTING

Cloud computing has emerged as a logical evolution of Information Technology (IT). It is driven by a worldwide increasing demand for utility priced, on-demand allocated services delivering software applications and IT infrastructure such as data storage. The demand and requirements for such services can, however, only be satisfied by leveraging economies of scale of IT resources, which is enabled by the technical advances in communication (e.g., the Internet), virtualization technologies, and mobile computing. Cloud computing merges the main advantages of its computing precursors, without the costly disadvantages associated with traditional data centers [11,38].

On-demand allocation of resources, utility pricing models, and online accessibility make the adoption of cloud computing ideal for start-up businesses, but even large enterprises are increasingly adopting it [36,38]. Cloud computing technologies and capabilities have enabled new application scenarios, and have also created additional user requirements that were not feasible previously.

New technologies influence both the architectures and the requirements of systems [2]. Thus, the architectures of ideal cloud hosted SaaS applications differ significantly from those of traditional enterprise applications. Consequently, new architectural principles and patterns have emerged, facilitating the IDEAL characteristics of cloud applications: *I*solated state, *D*istribution, *E*lasticity, *A*utomated management, and *L*oose coupling [11].

Many existing enterprise applications were developed decades ago, before the emergence of cloud computing. Due to their high development costs, there is a strong economic incentive to modernize and deploy them as worldwide on-demand accessible SaaS applications in cloud environments. As the architectures of these applications cannot be assumed to adhere to the IDEAL characteristics already, this requires reengineering and *architectural refactoring*, in order to fully take advantage of the new capabilities of cloud computing [44].

SaaS applications achieve a higher level of cost efficiency by sharing component and application instances between multiple organizations, thus eliminating the need to maintain and to deploy individual application instances for each organization. This concept is called *multitenancy*. A tenant is an organizational entity, usually a group of users, who rent access to a SaaS application [8]. Kabbedijk et al. define multitenancy as follows:

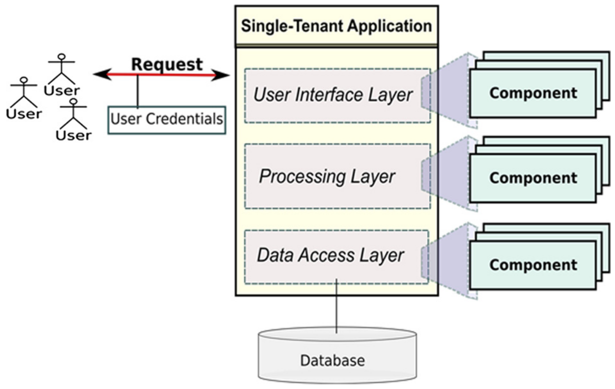

FIGURE 13.1

Single-tenant component based architecture

"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." [19]

Data security, confidentiality, and integrity are crucial in a multitenant environment [19]. The examples in this chapter show how architectural patterns can be applied to architecturally refactor and to reengineer a single-tenant legacy application (Fig. 13.1) into a modern, multitenant SaaS application (Fig. 13.2). Architectural refactoring refers to changing the internal structure of a system without changing its external behavior [43]. Reengineering, on the other hand, refers to changing the external behavior in order to meet new requirements. The transformation of a single-tenant legacy application into a multitenant SaaS application requires both architectural refactoring and reengineering on the code level. Although the application's external behavior after the transformation appears to remain unchanged from a tenant's (i.e., user's) point of view, it is changed significantly from a designer's and service provider's point of view.

This chapter focuses on how to avoid *data separation defects* that allow unauthorized tenants using the same application instance to access or modify other tenants' confidential data. Tenant data separation defects occur when data belonging to one tenant is made accessible to other, unauthorized tenants. This type of vulnerability could be caused either by software defects or by intentionally placed backdoors.

For a tenant t_i, let e_{ti} denote a data input into a shared component, and a_{ti} denote a data output from the component (i.e., for a tenant). Tenant data separation defects can occur, for example, in the following scenarios (Fig. 13.3):

(i) A service request input e_{t1} from tenant $t1$ (upper left corner) leaks into data storage output a_{t2} that affects a database entry belonging to tenant $t2$ (lower right corner). In this case, tenant $t1$ can modify tenant $t2$ data, resulting in a breach of data integrity for tenant $t2$. For example, this kind of data leak can occur if the data storage procedure writes data into the database using a different tenant ID from the one associated with the requesting tenant. This could happen by accident (e.g., conversion/casting error), or on purpose.

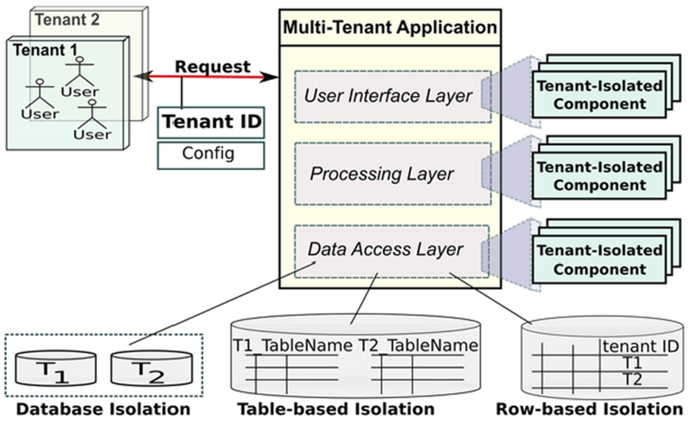

FIGURE 13.2

Multitenant architecture with tenant-isolated components

(ii) A data input from the database e_{t1} (database read operation) meant for tenant $t1$ (lower left corner) reaches a service response output a_{t2} meant for tenant $t2$ (upper right corner). In this case, tenant $t2$ can view tenant $t1$'s data, resulting in a breach of data confidentiality. This kind of data leak can be caused by a procedure that deliberately or accidentally reads data using a tenant ID that differs from the one associated with the requesting tenant.

(iii) Database entries from tenant $t1$'s data store (lower left corner) are copied into tenant $t2$'s data store (lower right corner), and later made accessible as belonging to tenant $t2$ (upper right corner). In this case, tenant $t2$ obtains read access to tenant $t1$'s data, and tenant $t2$'s data integrity is compromised. This is thus a breach of both data confidentiality and data integrity. This kind of data leak can occur, for example, in an application-to-application communication procedure that uses database integration (i.e., through shared database transfer tables). The sending component writes data into a shared transfer table, and triggers the receiving component to read data from the same table [5]. If the tenant ID is modified by either the reading or the writing component, a data leak occurs.

To prevent such data separation effects from happening and to safely architecturally refactor single-tenant applications for multitenancy SaaS Clouds, a *reengineering method* and supporting *architectural patterns* and *architectural refactorings* are required. The next three sections introduce such a method and patterns.

13.3 SOLUTION OVERVIEW: REENGINEERING METHOD AND PROCESS

The purpose of the reengineering method presented here is to assist architects who want to transform a typical single-tenant legacy enterprise application (Fig. 13.1) into a multitenant application that can be deployed in a Cloud environment (Fig. 13.2).

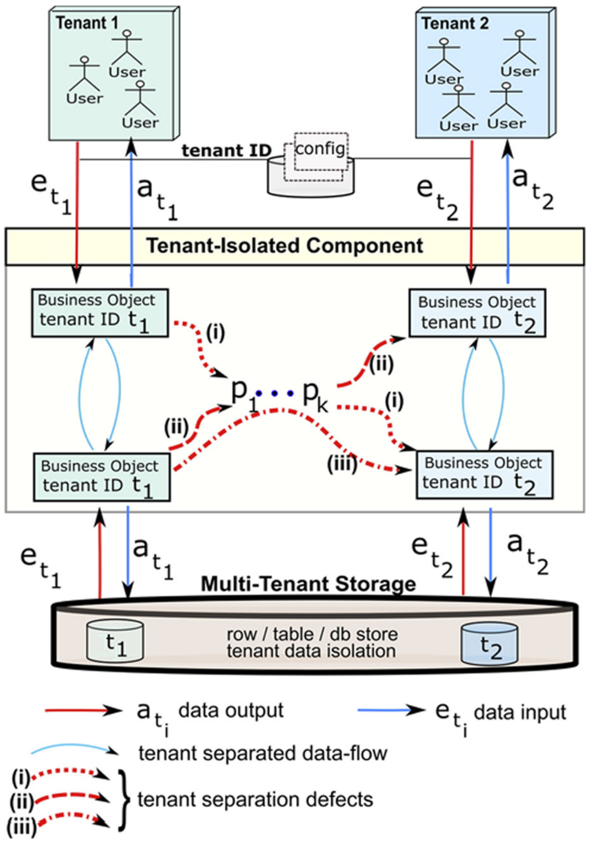

FIGURE 13.3

Illustration of data-flow defects causing data leaks in a multitenant, tenant-isolated component

Enterprise applications are comprehensive information systems that support complex business processes in and across business organizations. Although such applications are built for different purposes and therefore functionally very different, certain commonalities among them exist. For example, the architecture of typical enterprise applications consists of three tiers: a *resource tier* that provides access to a persistent database or other legacy resources, an *application tier* that implements the business logic, and a *presentation tier* that implements the presentation logic for clients and other integrated enterprise applications [10]. Further commonalities are related to how certain recurring software engineering problems are solved through the application of common solutions, so-called patterns. Such patterns are applied in enterprise systems to solve recurring problems related to the logical architecture [12], the integration of enterprise systems [17], or their deployment on cloud computing environments [11].

Aiming to cover a common type of legacy application, the focus here is specifically on Web application architectures that are based on the widely-used Model-View-Controller (MVC) pattern. Fig. 13.4 outlines a reengineering process that comprises three steps (that will be explained in Section 13.4.1),

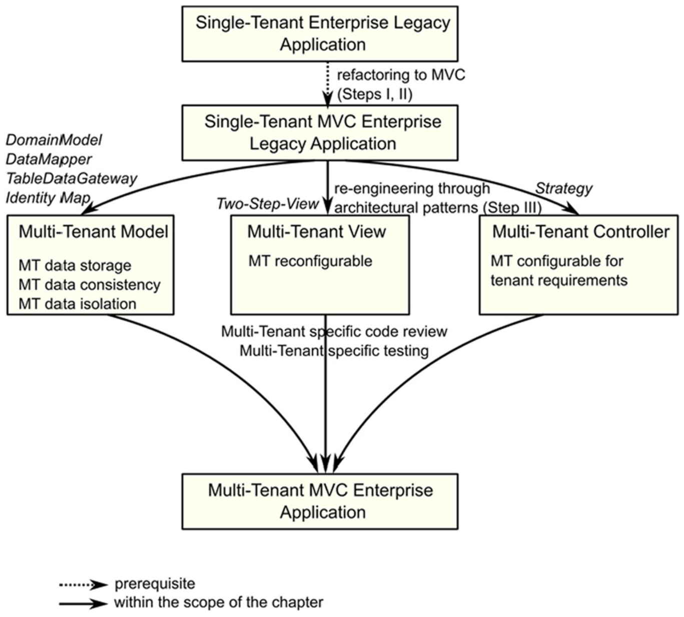
Single-Tenant Enterprise Legacy Application
refactoring to MVC (Steps I, II)
Single-Tenant MVC Enterprise Legacy Application
DomainModel
DataMapper
TableDataGateway
Identity Map
Two-Step-View
re-engineering through architectural patterns (Step III)
Strategy
Multi-Tenant Model
MT data storage
MT data consistency
MT data isolation
Multi-Tenant View
MT reconfigurable
Multi-Tenant Controller
MT configurable for tenant requirements
Multi-Tenant specific code review
Multi-Tenant specific testing
Multi-Tenant MVC Enterprise Application
prerequisite
within the scope of the chapter

FIGURE 13.4

The reengineering process for enabling multitenancy in a single-tenant legacy application by applying a combination of enterprise application architecture patterns [12]

each of which qualifies as an architectural refactoring according to Zimmermann [43]. Here it is assumed that the architecture of the legacy application is either already based on the MVC pattern, or that it can easily be refactored into a typical Web MVC architecture through standard pattern refactoring techniques [20,34].

Section 13.4 will elaborate on the multitenancy specific requirements for each of the MVC's components, along with architectural solutions using enterprise application architecture patterns. Thus, the resulting architecture is multitenancy capable, and fulfills all architectural requirements that are specific to multitenancy.

At the code level, these requirements, such as tenant data separation, need to be verified through code reviews and regression testing techniques. These techniques will be explained in detail in Section 13.5, using a PHP music library application as an example.

13.4 SOLUTION DETAIL 1: ARCHITECTURAL PATTERNS IN THE METHOD

This section elaborates on the typical architectural requirements that emerge from the new capabilities of cloud environments, and introduces architectural patterns that can be applied to build multitenancy capable systems. First, the typical steps required for modernizing legacy applications for cloud environments are described as architectural refactorings.

At the component level, the new architectural requirements for such applications are identified and architectural patterns are selected that are suitable to exploit the technological benefits and capabilities of cloud environments. Finally, a possible architectural solution is shown by applying a series of suitable architectural patterns that lead to the new envisioned multitenant architecture.

13.4.1 ARCHITECTURAL REENGINEERING STEPS FOR THE CLOUD (ARCHITECTURAL REFACTORING)

A good architecture exploits the technical advantages and capabilities of the target environment, while satisfying the functional and nonfunctional requirements of the system's stakeholders. However, there are additional characteristics, that make an architecture better than "good enough", i.e., "beautiful" [33]. The characteristics that make a "beautiful" architecture stand out from a "good enough" one are support for incremental construction, the ability to easily support changes (persistence), ease of use for both developers and users, and a conceptual integrity that makes it easy to learn [33]. The process described here aims to achieve these objectives.

In this case, a cloud computing environment is the target. The typical properties of cloud environments are (i) on-demand self-service, (ii) broad network access, (iii) measured service, (iv) resource pooling, and (v) rapid elasticity [11,24]. Requirements (i) to (iii) do not directly influence architectural decisions in a multitenancy context. Therefore, the focus is here in particular on architectural decisions that enable *resource pooling* and *rapid elasticity*.

Rapid elasticity refers to the system's ability to quickly allocate additional resources when the workload increases, and to de-allocate them when the workload decreases. Ideally, this capability is supported by the application. To achieve this, the application is scaled horizontally (i.e., scaled-out), by starting up additional service instances that take up the additional workload. These on-demand service instances can be shut down automatically when the workload decreases. An ideal architecture for SaaS applications is therefore service-oriented, and supports dynamic workload balancing across loosely coupled, stateless services. The MVC pattern and modern MVC-based presentation layer frameworks adhere to these characteristics (which, in a reengineering process, can be achieved by Step II in Fig. 13.4).

Resource pooling refers to the system's ability to allow multiple customers (i.e., tenants) to share the allocated resources. At the application level, this means sharing components or application instances, in other words, to enable multitenancy. Ideally, the fact that resources are shared should not be noticed by the tenants. A tenant should not be able to distinguish a multitenant component or application from one that is exclusively dedicated to it. Multitenancy is achieved by specifically designing the architecture to meet the specific multitenancy requirements. The following subsection elaborates these requirements and discusses pattern-based solutions.

From an architectural perspective, as motivated in Section 13.2, cloud applications should expose IDEAL architectural characteristics: isolated state (i.e., stateless components), distribution (i.e., SOA), loose coupling, elasticity (i.e., on-demand allocation and de-allocation), and automated management [11].

To achieve these characteristics, the following architectural reengineering steps are typically required (Fig. 13.4):

(i) Step I: Identify and define the legacy application's functional layers in terms of data access, data processing, and user interface/service interfaces.

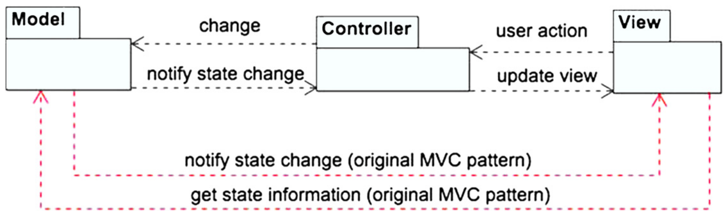
Model
change
Controller
user action
View
notify state change
update view
notify state change (original MVC pattern)
get state information (original MVC pattern)

FIGURE 13.5

The Web MVC architectural pattern

(ii) Step II: Decompose each layer into loosely coupled, stateless components that can be deployed as on-demand services elastically.

(iii) Step III: Reengineer the components to support multitenancy. This step is described in detail below, while Steps I and II are beyond the scope of this chapter.

13.4.2 MULTITENANCY REQUIREMENTS AND PATTERNS FOR CLOUD ENVIRONMENTS

The logical architecture of typical enterprise applications is typically structured in three layers (Fig. 13.1): the data access layer, the processing layer, and the presentation layer. This layered structure can also be observed in the Model-View-Controller (MVC) pattern.[1] Therefore, and as already motivated in Section 13.3, it can be assumed that the architecture of the legacy application either resembles MVC already or that it can be first reengineered into a Web MVC application with existing refactoring techniques [20,34].

The MVC pattern is an architectural pattern that leads to a functional decomposition into three groups of components: the Model, the View, and the Controller. The Model components represent application data and data access layer, the View components render data for the user, and the Controller components handle user inputs and manipulate both views and models.

The majority of modern Web applications are based on this pattern; however, most Web application frameworks (e.g., the PHP Zend Framework) implement a slightly modified version of the MVC, the so-called “Web MVC” or “Model 2” pattern (Fig. 13.5) [3,7]. In the original MVC pattern, the View is an Observer of the Model, and thus receives direct notifications when changes in the Model’s data occur. On the other hand, in the Web MVC version, these notifications are redirected through the Controller instead, as this is easier to realize with many Web technologies (Fig. 13.5).

The specific requirements which have been identified for multitenancy capable MVC components are explained below [8,39].

[1]Two different conceptions exist regarding the MVC pattern: (i) applied at the presentation layer only [12], or (ii) applied across a layered architecture, to separate the presentation layer from the data modeling of the business layer [40]. In this chapter its application across layers is assumed.

13.4.3 THE MULTITENANCY CAPABLE MODEL

Requirement R1: A major requirement for a multitenant MVC Model is its ability to store application domain objects in a multitenant database and to map in-memory domain objects to a multitenant relational database schema. Three database solutions are common for multitenancy (Fig. 13.2) [13]:

(i) A dedicated database instance for each tenant,
(ii) A shared database instance, with a dedicated set of data tables for each tenant, or
(iii) A shared schema in a shared database instance with a tenant ID associating each row to its tenant owner.

An ideal MVC Model should support all three multitenant database options, and also hybrid solutions with a combination of the three options.

Solution for R1: A pattern-based solution is to combine the patterns “Data Mapper” and “Table Data Gateway” [12]. The “Data Mapper” pattern maps in-memory objects to one or multiple relational databases, while the “Table Data Gateway” pattern provides a simple interface for storing and retrieving data from individual database tables (Fig. 13.6). The “Domain Model” pattern [12] combines behavior and data, and is a common pattern in enterprise applications with complex logic [12,41]. In this chapter the domain object is associated with the Tenant class through the tenant ID property. The combination of these patterns decouples the Model from the underlying database technology, allowing the integration of various databases in support of multitenancy.

Requirement R2: The appropriate separation of tenant data is crucial for a multitenant application, and domain objects belonging to different tenants have to be linked to a tenant identifier.

Solution for R2: The “Identity Field” pattern [12] can be used to maintain the identity between the database rows and in-memory domain objects. Each domain object is assigned a tenant ID, which is mapped to the database tenant identifier used to link the data entry with the tenant owner (Fig. 13.6).

Requirement R3: In order to ensure data consistency and separation, a one-to-one relationship between tenant database entries and in-memory domain objects needs to be enforced, and access to tenant domain objects has to be controlled and restricted. Each tenant should be able to access only its own data. At the database level, this can be enforced by ensuring that a tenant ID parameter is mandatory when accessing the database. If a database query ignores the tenant ID parameter, it could access all tenants’ data. Such an attempt to access all tenants’ data can be discovered at run-time, for example, by introducing a “dummy or sentinel value” that triggers an error when it is accessed [28].

Solution for R3: The “Identity Map” [12] pattern can be applied to avoid reloading identical database rows into different in-memory domain objects. When applying this pattern, all domain objects are loaded and stored in a common data structure, the “Identity Map.” Consequently, access to all domain objects can be controlled and restricted by the “Identity Map.” Read/write access is only granted to model objects owned by the accessing tenant (Fig. 13.6).

13.4.4 THE MULTITENANCY CAPABLE CONTROLLER

Requirement R4: The Controller handles user input and manipulates data in MVC views and MVC models. Ideally, these functionalities are configurable for each individual tenant. Each tenant should be able to configure the application to handle input, store data, and display the information according to their own business requirements.

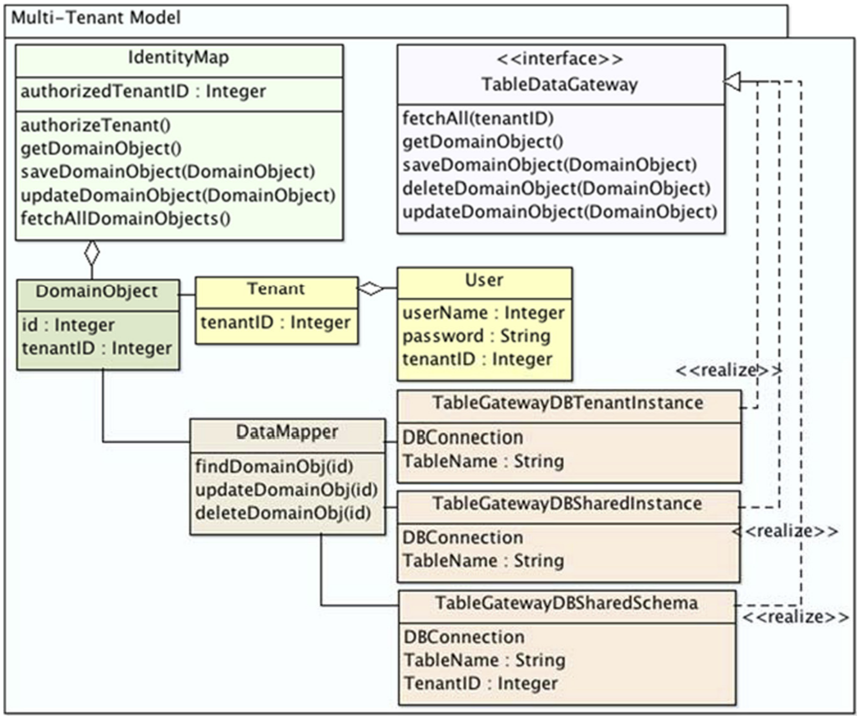

FIGURE 13.6

Multitenant MVC model based on the patterns Identity Map, Domain Model, Identity Field, Data Mapper, and Table Data Gateway

Solution for R4: The "Strategy" pattern [29] allows decoupling the behavior implementation (strategy) from an object, and therefore facilitates the implementation of an individual controller behavior for each tenant (Fig. 13.7).

In single-tenant Web applications, two options are typical for the implementation of controllers: (i) one Controller–multiple Views, and (ii) multiple Controllers–multiple Views (i.e., one Controller per View) [14]. The multitenant equivalents are:

(i) One Controller per tenant – a group of multiple Views per tenant, and
(ii) A group of multiple Controllers per tenant – a group of multiple Views per tenant (i.e., each tenant-specific controller handles one tenant-specific view).

13.4.5 THE MULTITENANCY CAPABLE VIEW

Requirement R5: An ideal multitenant application should allow tenants to individually configure the layout of data and the user controls displayed in HTML pages.
Solution for R5: The "Two-Step-View" (Fig. 13.8) pattern allows presentation layer architects to decouple the displayed information from the layout. In the first stage, this pattern forms a logical page,

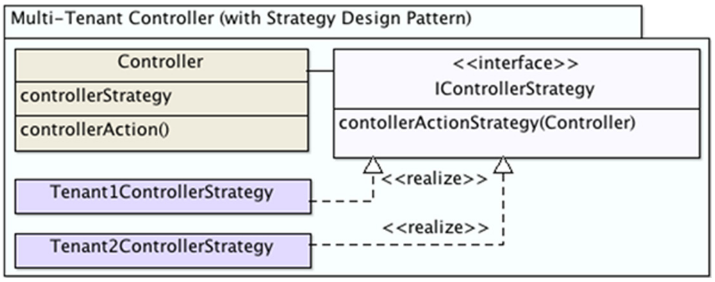
Multi–Tenant Controller (with Strategy Design Pattern)
Controller
controllerStrategy
controllerAction()
<<interface>>
IControllerStrategy
contollerActionStrategy(Controller)
<<realize>>
<<realize>>
Tenant1ControllerStrategy
Tenant2ControllerStrategy

FIGURE 13.7

Multitenant MVC controller based on the strategy pattern

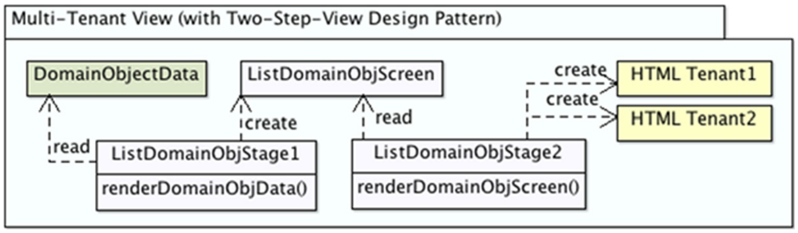

FIGURE 13.8

Multitenant MVC view based on the two-step-view pattern

which is rendered into HTML in a second stage. While the first stage can be application-wide, the second step, which influences the appearance, is individually configurable for tenants (see Section 13.6, for example). The individual tenant-specific creation of HTML pages depends on the tenant ID properties stored in the Domain Object data.

13.5 SOLUTION DETAIL 2: TESTING AND CODE REVIEWS

The pattern-based approach described above enables a developer to create a multitenant architecture. However, the corresponding implementation of this architecture still has to be evaluated for correctness. Most importantly for a multitenant application, it is necessary to ensure that the appropriate separation of tenant data is preserved at the code level. This can be achieved with testing and code review techniques, as explained in this section.

13.5.1 TESTING FOR MULTITENANCY DEFECTS

Enterprise Web applications are difficult to test thoroughly because they often consist of multiple and complex multistage processes and backend system integration flows (that are hard to mock/simulate), and are simultaneously accessed by multiple users [15,25,32]. In such applications that require

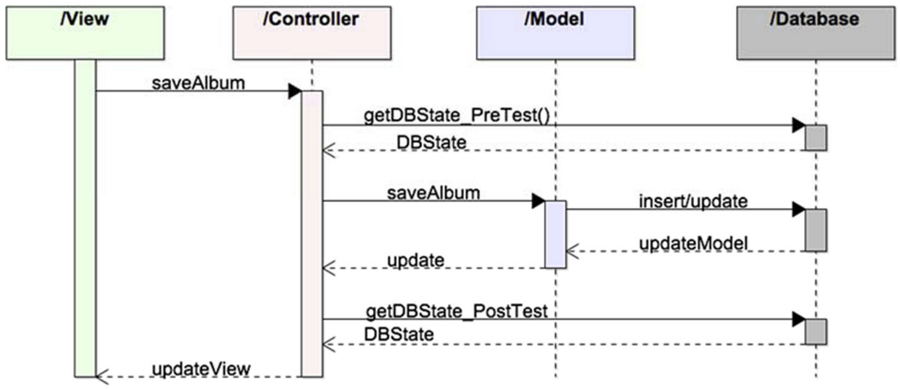

FIGURE 13.9

Sequence diagram of a testing procedure for multitenancy defects

security-testing, tenant data separation defects are especially difficult to find because security leaks may manifest themselves as side-effects on other tenants' data. Such defects can remain undetected by functional black-box testing techniques [35]. The example below illustrates one such tenant data separation defect. Subsequently, a solution using a white-box testing technique is presented.

Example: Confidentiality and Integrity Breach. In this example a multitenant Web application allows users to edit entries of a music album library. Changes made by the user are correctly reflected in the model and in the database (Fig. 13.9). However, in addition to this correct behavior, assume that a software defect allows some edited entries to be duplicated, associated with a different tenant ID, and thus stored in another tenant's data. This software defect results in a breach of data confidentiality for the first (editing) tenant, and a breach of data integrity for the second tenant (as motivated in Fig. 13.3). Such breaches can have a significant negative impact on the business, e.g., harm an application's audit compliance [18].

Since this program completes its intended database update correctly, a typical unit test to ensure that the specific database row has been changed as required will not reveal any problems. Additionally, therefore, a test is needed which checks for the unexpected side-effect that this code has on another tenant's data. In this test the state of the database is retrieved before and after the execution of the program functionality under test (Fig. 13.9). In this example, prior to executing the "saveAlbum" function, the number of rows per tenant is obtained from the database table, and the test ensures that the table is not empty. After executing the "saveAlbum" function, the test obtains again the number of rows, and ensures that it is unchanged (Listing 1).

It is important to highlight that the states of the database are obtained directly from the database, bypassing the Model. These database states consider the data rows and tables of *all* tenants before and after executing a particular program function. The test then asserts, by comparing the states before and after the function, that the tenant data separation property is maintained. In order to assert the tenant data separation property, the database states need to take into account the number of table rows, as well as checksums of all rows (of all tenants) expected to remain unaffected.

```
void testSaveAlbum {
    int rowsBefore = getNoOfRows(AlbumTable);
    assert (numberOfRows > 0);
    editAlbum(..); //function under test
    int rowsAfter = getNumberOfRows(AlbumTable);
    assert(rowsBefore == rowsAfter);
}
```

Listing 1: Pseudocode of the test procedure

The test then asserts that only a single row belonging to the intended tenant is updated, while the checksums (and therefore content) of all other rows remained unaffected. To avoid interference from other updates, this type of test can only run with a single logged-in user. This type of testing should be performed with a small testing database, since a very large multitenant production database would negatively affect the execution performance of the test. This test should be disabled in the production code.

13.5.2 CODE REVIEW FOR MULTITENANCY DEFECTS

Testing often does not reveal all defects or vulnerabilities. For example, a back-door or software defect which allows one particular tenant (with a secret tenant ID) to access all other tenants' data would very likely remain undetected by tests, especially if the privileged tenant had not yet been created at the time of testing. Nevertheless, such tenant data separation defects can be revealed by code reviews.

It is important to note that tenant data separation defects can occur anywhere along data-flow paths between application inputs and outputs (Fig. 13.3 in Section 13.2). Thus, the code reviewing process for finding tenant data separation defects differs from the process for finding security vulnerabilities, which is typically limited to analyzing input and output procedures [9]. To ensure appropriate data separation in a multitenant application, the code reviewer needs to identify and analyze all data-flow paths of input-value/tenant-identifier pairs leading to output-value/tenant-identifier pairs, and ensure that no data output is produced for a different tenant.

For instance, in the code example, the Zend "skeleton" music application, the program was implemented using the "Domain Model" pattern [12]; a common pattern in enterprise applications. Therefore, all objects containing tenant-specific data used as application input or output were encapsulated in classes derived from such a common Domain Object base class. The Domain Object base class and thus all derived subclasses inherit a pair property (data-value, tenant-id) that associates each data value with the tenant owner.

Tenant data separation defects can occur when a multitenant program that receives an input domain object belonging to a tenant, produces at least one output domain object which belongs to a different tenant. In order for this to occur at runtime, at least one internal function needs to violate the tenant data separation property.

The association between tenant data and tenant identifier can be violated in a domain object either by (i) overwriting the tenant identifier with a different one, or (ii) by overwriting the domain object data with other data that belongs to a different tenant. The data-flow of such corrupted domain objects needs to be analyzed, in order to determine whether they can reach an application output.

The code reviewing process needed for finding tenant data separation defects can be summarized as follows:

(i) Identify all procedure invocations which allow data inputs and outputs. Such procedures include, for example, user interfaces (MVC Views), but also database and file system access procedures (MVC Models).

(ii) Find connecting paths of data-flow leading from application inputs to application outputs. Data-flow can occur explicitly, through variable assignments, parameter passing to procedures, and inter-procedural through assignments to and from global and static variables. Data-flow can also occur implicitly through control dependencies, e.g., in code segment "if ($x ==$ false) then $y = 0$" the value of variable y is influenced by that of variable x even though there is no assignment between them [27].

(iii) For each identified data-flow path, verify through manual code-reviews that no data-flow exists from input-value/tenant-identifier pairs to a different tenant.

13.5.3 SUMMARY

Table 13.1 summarizes the method and populates the architectural refactoring template from Table A.1 in Appendix 13.A with the multitenancy reengineering knowledge from Sections 13.3, 13.4, and 13.5.

13.6 CASE STUDY (IMPLEMENTATION)

This section demonstrates the applicability of the above reengineering method and pattern-based cloud application architectures with a fictitious, but still representative, sample enterprise application. The method is applied to a typical PHP Web application, which is incrementally transformed from a single-user application into a multitenant SaaS application using two different techniques. The first reengineering technique aims at keeping the number of source code and architecture changes to a minimum, and ignores the use of patterns.

The second reengineering technique that is applied is the one introduced in Section 13.4. The evaluation of the two reengineering techniques confirms that the pattern-based architecture is better suitable to ensure tenant data separation.

The case study application is based on the Zend Framework Skeleton Application [42], and it is a single-user PHP Web application. A PHP case study is used because PHP has evolved into a powerful object-oriented language [23], and is currently one of the most popular server-side programming languages, used, for example, by Facebook, Baidu, Wikipedia, and Twitter [37].

The Zend Skeleton Application is a standard example that demonstrates PHP's capabilities for enterprise applications, and therefore an ideal candidate for our purpose. The Zend Skeleton Application is a music library management system that allows users to enter, edit, delete, and list music albums in a library (Fig. 13.10).

Table 13.1 Architectural refactorings to introduce multitenancy support

Architectural Refactoring (AR)	**Introduce multitenancy support to layered application.**
Context	An application already applies logical layering and the Model-View-Controller (MVC) pattern on a conceptual level; it is supposed to be deployed into a cloud that has a shared usage model and pools computing and storage resources.
Stakeholder concerns and quality attributes (design forces)	• Security (data privacy, data confidentiality, data integrity) • Accuracy (of calculations and data storage) • Performance • Maintainability • Cost (of deployment and hosting)
Architectural smell	A cloud application has been designed for single-tenant use and therefore does not provide any protection against data separation defects.
Architectural decision(s)	• Cloud deployment model, cloud service model • Approach to tenant isolation • Detailed design of components in MVC pattern
Evolution outline	Upgrade single-user/-tenant component architecture into a multitenant, patterns-based architecture by enhancing the MVC usage with: 1. A Multitenancy Capable Model (with tenant identity-aware domain objects), 2. A Multitenant Two-Step View, and 3. A Multitenancy Capable Controller (strategy-based) as outlined in in Section 13.2 (in Fig. 13.2) and in Section 13.3 (in Fig. 13.4).
Affected architectural elements	All three top-level components in the MVC pattern, as well as the components in Fowler's supporting patterns discussed in Section 13.4 and listed in Table 13.3 in Section 13.6 (e.g., Identity Map, Two-Step View, Strategy).
Execution tasks	1. Implement the enhanced patterns in chosen presentation layer, business logic layer, and data access layer technologies (possibly supported by frameworks). 2. Define tenant identifiers. 3. Update database schemes and system configuration means. 4. Test for multitenancy defects. 5. Review code to validate correct implementation of multitenancy support. 6. Perform security compliance audit (w.r.t. tenant data separation). 7. Monitor application performance at runtime.

13.6.1 MULTITENANCY TRANSFORMATION WITHOUT PATTERNS

The goal of this first transformation is to implement multitenancy with as few code and architecture changes as possible. Starting with the original application (Fig. 13.10), this can be achieved as follows (Fig. 13.11 and Table 13.3).

First, a tenant ID property is added to all domain classes, i.e., in this case the Album class, allowing us to associate each album object with a tenant owner. A new User class includes a tenant ID property through which it associates each user with a tenant. The new authentication controller and the corresponding login view allow users to log-in. The modified Table Gateway class restricts the access

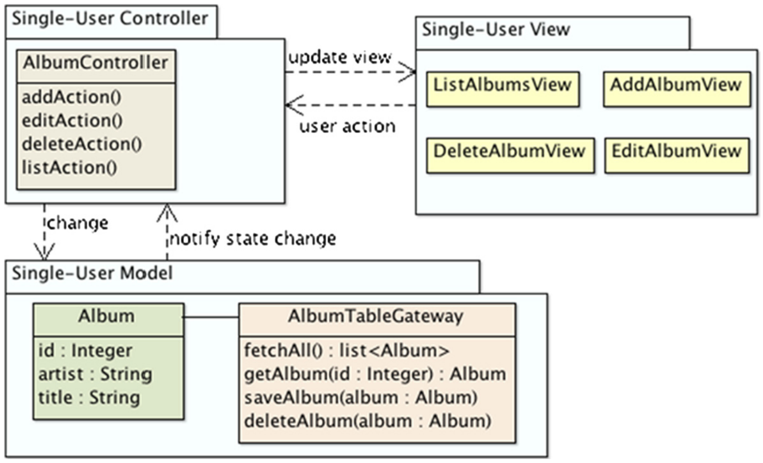

FIGURE 13.10

MVC architecture of the music library management system

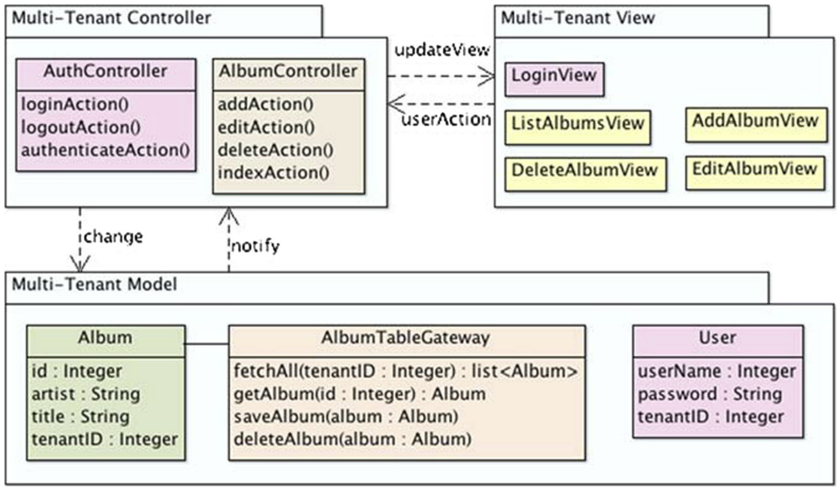

FIGURE 13.11

Reengineering without architectural patterns

to entries associated with the specified tenant. In the database, an additional tenant ID column links each entry to the tenant owner (in a shared multitenant database schema). Authorization checks are implemented as part of the controller actions.

13.6.2 MULTITENANCY TRANSFORMATION WITH PATTERNS

A second version of the Zend Skeleton application can also be produced; however, this time by applying the reengineering method established in Section 13.3 and the architectural patterns introduced in Section 13.4.

The transformation steps in this second version are as follows (Fig. 13.12). In the MVC Model, the "Data Mapper" pattern in implemented in addition to the already existing "Table Data Gateway pattern." This results in a new "Two-Level Data Mapping Gateway" pattern. The first level ("Album Data Mapper") exposes functionalities for retrieving, updating, and deleting an Album object from the Model, while the second level, consisting of the particular realizations of the "Table Data Gateway" interface, implements the data-level read/write access to different types of multitenant databases. Therefore, these patterns facilitate the integration of multitenant databases with database isolation, table-based isolation, or row-based isolation schemas (Fig. 13.2).

The next step is to add the "Identity Map" pattern to the Model, as well as the "Domain Model" with an Identity Field, to form an "Identity-Aware Domain Model" pattern. Through the combination of these patterns, the Model encapsulates the functionality for creating, retrieving, updating, and deleting domain objects in a single class, namely the "Identity Map." In other words, the "Identity Map" is the only access point for data input and output to and from the Controller. Therefore, since the code is not spread out over multiple classes, it is easier to test and review in order to ensure that the Model maintains appropriate tenant data isolation.

In the Controller, the Strategy pattern is implemented with different controller strategies for different tenants. The resulting loose coupling between the strategy implementations and the application's "Album Controller" through the interface "IAlbumControllerStrategy", makes it is easier to add, remove, or modify tenant-specific controller strategy implementations without affecting the rest of the code base. Individual implementations of controller strategies that meet a particular tenant's requirements do not affect other tenants' controller functionalities. The same holds for any software defects in these tenant-specific strategies.

Finally, each MVC View page is transformed into a "Two-Step-View." This decouples the data content from its presentation layout, and allows therefore implementing tenant-specific and individually configurable user interfaces for each tenant. Fig. 13.12 and Table 13.3 summarize the architectural refactoring process.

13.6.3 COMPARISON

The first, patternless transformation version shows that multitenancy functionality can be achieved easily with very few changes of the architecture and source code. However, although the resulting application could be used by multiple tenants, it does not meet the requirements of a multitenant application, including the IDEAL cloud application architecture principles (Table 13.4). This first version is more difficult to integrate with different multitenant database schemas, because the Album Table Gateway is only able to access a single database schema. Additionally, since the MVC Controller and the MVC View are designed for single-tenant use, the implementation is not sufficiently flexible to be easily customized for individual tenant needs. Most importantly, since the source code contains tenant authorization checks at different places (e.g., in each Controller action), it is more difficult to test and code review in order to eliminate tenant data separation defects.

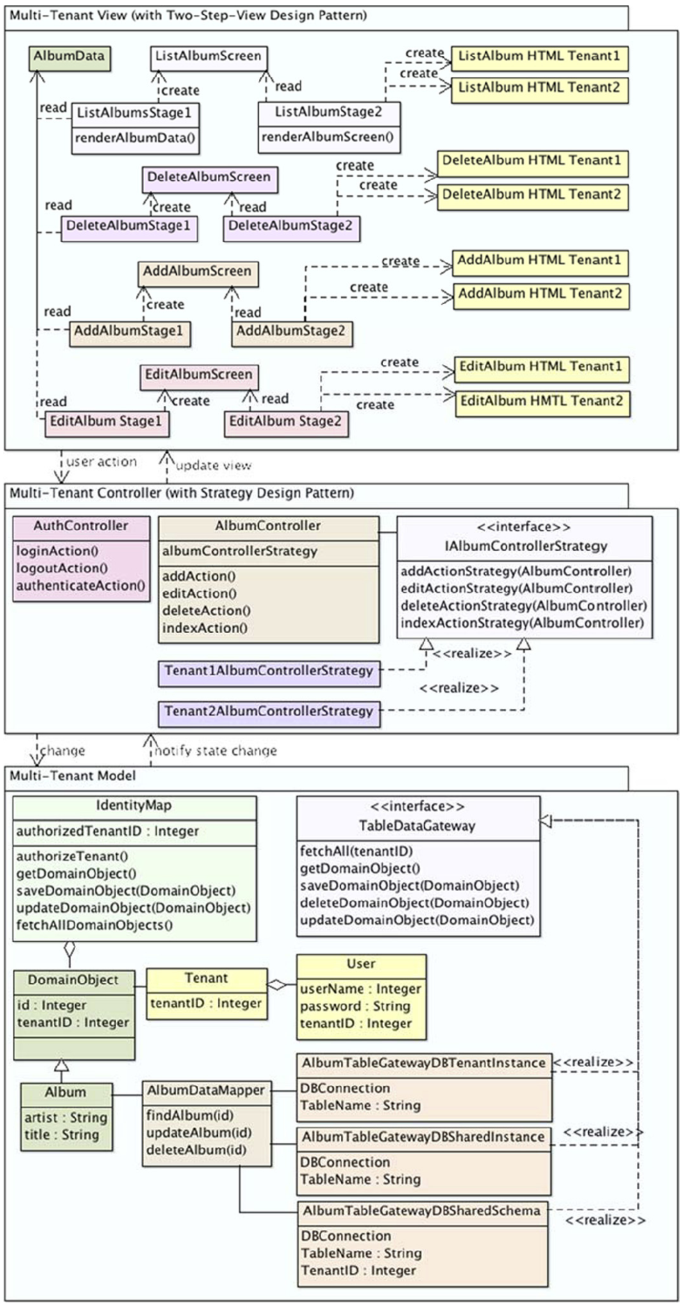

FIGURE 13.12

Pattern-based multitenant capable MVC architecture

Table 13.2 Number of classes and size of the case study application in its transformation stages from a single-user application to the pattern-based version of the multitenant architecture

Application transformation stage	Classes/PHP files	Lines of code
Original Zend Skeleton Application (Modules only, without Zend Framework)	10	561
Multiuser (single-tenant)	18	1021
V1: Multitenant without patterns	18	1052
V2: Multitenant with patterns	28	1344

Table 13.3 Summary of reengineering steps (architectural refactorings). The patterns are introduced by Fowler [12]

	Model	View	Controller
V1: Multitenant without patterns	The tenant ID property is added to class Album	N/A: not tenant-specific configurable	Controller retrieves the tenant ID of the authenticated user. All Controller actions use the tenant ID.
V2: Multitenant with patterns	Added classes: User, Tenant, DomainObject (abstract base class), AlbumDataMapper, IdentityMap, interface ITableDataGateway	Refactored to Two-Step-View (supported by Zend Framework)	Refactored to Strategy pattern. Added interface IAlbumControllerStrategy implemented by tenant-specific AlbumControllers

Table 13.4 Evaluation with respect to "IDEAL" Cloud application characteristics

	Isolated state	Distribution	Elasticity	Automated management	Loose coupling
V1: Multitenant without patterns	Good (stateless components)	Suitable (MVC property)	No (not suitable for horizontal scaling)	No	Weak
V2: Multitenant with patterns	Good (stateless components)	Suitable (MVC property)	Yes (horizontal scaling)	Yes (configurable Controller and View)	Yes

On the other hand, the second, pattern-based version (Fig. 13.12) requires more code and architectural changes (Table 13.2). However, the resulting architecture is superior with respect to its ability for incremental construction, the ability to support changes, and its conceptual integrity. For example, the incremental construction is supported by allowing the addition of new tenant-specific customized controller strategies and Views, without affecting the implementations of other tenant's Controllers and Views. Changes made in such tenant-specific classes do not affect the entire source code base, and are therefore easier to test, debug, and deploy. Furthermore, the use of mature enterprise application architecture patterns improves the conceptual integrity and recognizability of the entire architecture.

Table 13.5 Evaluation with respect to multitenancy requirements

Requirement	V1: Multitenant without patterns	V2: Multitenant with patterns
R1 (multitenant storage)	No	Yes
R2 (data separation)	Ad hoc solution	Yes
R3 (limit data access)	No	Yes
R4 (configurable controller)	No	Yes
R5 (configurable view)	No	Yes

The internal coupling properties of the MVC components remain unaffected by the application of the new architectural patterns. The MVC pattern already achieves loose coupling characteristics of its three major components. However, the multitenancy capable MVC Model is more decoupled from the underlying database.

13.7 DISCUSSION

As evidenced in Table 13.5, the pattern-based version of the refactored music library management system fulfills the multitenancy requirements completely, while the nonpattern based version does not do so. Furthermore, the pattern-based version is easier to adapt to different cloud offerings. For example, the loose coupling between the model and the underlying data storage enables the simultaneous integration with multiple cloud storage offerings (Fig. 13.2). Its processing functionalities and user interfaces are easier to configure to individual tenant-specific needs: for instance, the Multitenancy Capable Controller can be configured by allowing each tenant to implement an individual strategy (Fig. 13.7), while the user interface can be configured to display a customizable layout for each individual tenant (Fig. 13.8).

The Multitenancy Capable MVC Model also simplifies the checks of correctness with respect to tenant data separation properties because pattern instances are rather easy to locate in code (assuming that its development adheres to naming conventions and/or uses architecturally evident coding styles). Regression tests, and code reviews can be focused on ensuring that Domain Objects are assigned the correct tenant identifier at the time of construction. Later changes of the tenant identifier can be prevented through code, as well as the cloning of Domain Objects with a different tenant identifier.

The Identity Map is the only interface for passing Domain Objects from and into the model. Therefore, authorization checks implemented in the Identity Map can ensure that each tenant is only granted access to its own Domain Objects. Correctly implemented, these mechanisms guarantee that the tenant data separation property is maintained by the application. Due to the complexity of the required changes and their impact on the entire application, the reengineering process is manual, and would be difficult to automate.

The case study presented in Section 13.6 showed that tenant data separation defects are more likely to happen and are more difficult to detect in the first version (which was developed in an ad hoc manner), in comparison to the second, pattern-based version (whose implementation followed the reengineering method from Section 13.3). This is due to the fact that the data-flow graph of tenant-specific data is significantly simpler in the pattern-based version. On the other hand, the complex data-flow in the ad hoc version leads to difficulties in detecting programming errors.

13.8 RELATED WORK

The method described in this chapter concerns the reengineering of legacy applications to support multitenancy, with a specific concern for tenant data separation. Prior work has focused on reengineering legacy applications for cloud migration; however, few publications exist regarding the application of architectural patterns for this purpose. Xuesong et al., for example, proposed guidelines for reengineering legacy Web applications for multitenancy, but they considered only how to deal with the Singleton design pattern in multitenant applications [39]. They did not consider the more commonly-used MVC pattern as done above.

Kabbedijk et al. recently reviewed multitenant understanding in academia and industry, and their definition of multitenant systems was assumed in this chapter [19]. Most interestingly, after studying 761 research papers and 371 industrial blogs, they conclude:

> *"The results show that most papers propose a solution related to multi-tenancy, but almost no papers report on industrial experiences while implementing multi-tenancy, providing some insight into the maturity of the domain." [19]*

Earlier, Bezemer et al. described the challenges of reengineering for multitenancy [8]. However, while they recognized the ability to configure SaaS applications for tenant specific needs, this topic was left open. Similarly, Kousiouris et al. explained the multitenancy challenges of transforming legacy applications, mostly focusing on performance issues [21].

Sengupta et al. discussed the remaining research challenges related to SaaS applications, such as design, testing, maintenance, and optimization [30]. Interestingly, they also highlight refactoring from single-tenant applications as one of the challenges, which is one of the topics addressed in this chapter.

Kun et al. proposed a completely different approach to reengineering source code. They described a middleware layer which intercepts SQL database queries and converts them into multitenant database requests [22]. However, it is not clear from their paper whether their proposal has been implemented or not.

13.9 SUMMARY AND CONCLUSIONS

This chapter presented a pattern-based reengineering and architectural refactoring method for transforming legacy enterprise applications designed for single-tenant access into multitenant applications that satisfy the architectural principles required for successful SaaS deployment in cloud environments.

The appropriate separation of tenant data is one of the most critical properties of multi-tenant applications that process sensitive enterprise data because it directly affects the success or failure of SaaS solutions; data separation violations can have disastrous economic and legal consequences for both service consumers and service providers. As a minimum, multitenant SaaS applications should pass a compliance audit to gain evidence and confidence that enterprise data will not be accessible by competitors using the same SaaS solution (in addition to being protected against external attacks). Therefore, the reengineering of legacy applications and the development of new multitenant SaaS solutions should prioritize this critical requirement to preserve tenant data separation on all application and data storage levels. Ideally, the application architecture should make it easier for developers, testers, and compliance auditors to confirm this property, for example, through focussed testing and code reviewing. In this chapter, we presented a method and patterns for doing so.

The multitenancy aware architecture presented in this chapter extends existing enterprise application architecture patterns on the three logical architectural layers (i.e., user interface, business logic processing, and data access) reflected in the Model-View-Controller (MVC) pattern into multitenancy-enabled variants that satisfy five multitenancy-specific requirements. Most importantly, the presented approach preserves the tenant data separation property in order to guarantee data integrity and data confidentiality of sensitive tenant user and application data. Experience with this approach shows that the resulting multitenant software architecture and supporting patterns-based reengineering and architectural refactoring method are better suited for a systematic analysis and testing for the presence of tenant separation defects than ad-hoc evolutions of existing architectures.

A practical case study using a common framework for enterprise PHP applications, the PHP Zend Framework, confirmed the viability and the advantages of the approach. Both an ad-hoc and a pattern-based version of a music library management system were implemented and the two implementations were compared with respect to quality attributes such as developer productivity, maintenance effort, and IDEAL cloud application properties, as well as specific multitenancy SaaS requirements. This evaluation demonstrated that the investment into a pattern-based and method-supported approach to achieving multitenancy can pay off because it reduces the risk of data privacy/integrity breaches (among other security threats) and also simplifies maintenance, compliance auditing, and further evolution of the cloud application (i.e., SaaS offering).

The scope of this chapter was limited to static (architectural) application aspects, and did not include other equally relevant aspects of multitenant systems, such as performance, availability, reliability and fault-tolerance [16], or scalability, load balancing, or RESTful microservices [26]. Future work concerns improving the practical adoption of pattern-oriented architecting and architectural refactoring through Web-enabled architectural knowledge management systems for pattern-based solutions [44], that can be integrated into other tools used by software architects during initial design as well as maintenance and evolution, for instance, modeling and architectural decision management tools [45].

APPENDIX 13.A ARCHITECTURAL REFACTORING (AR) REFERENCE

The introduction of tenant-aware patterns in the three reengineering steps in Fig. 13.4 (i.e., Multitenant Model, Multitenant View, Multitenant Controller) qualifies as an architectural refactoring [44]. Such architectural refactorings can be compiled, curated, and shared in an Architectural Refactoring Tool (ART) [4]. Table A.1 shows the architectural refactoring template underlying Table 13.1 in Section 13.5, which was established in our previous work [43] and is supported by ART.

Table A.1 Decision- and task-centric Architectural Refactoring (AR) template (adapted from [43])

AR name	How can the AR be recognized and referenced easily?
Context	Where (and under which circumstances) is this AR eligible? The context section may include information about the viewpoint and/or abstraction/refinement level in an enterprise architecture management framework.
Stakeholder concerns and quality attributes (design forces)	Which nonfunctional requirements and constraints are affected by this AR?
Architectural smell	When and why should this AR be considered?
Architectural decision(s)	Applying an AR implies revisiting one or more architectural decisions; which ones? Such decisions may deal with pattern, technology, and platform choices.
Evolution outline	Which design elements does the AR comprise of? This is the center piece of the AR, providing a solution sketch. Since the AR describes a design change, two solution sketches may be provided (one illustrating the design before the AR is applied, one the design resulting from the application of the AR). Pitfalls to avoid might also be listed here (e.g., overdoing it).
Affected architectural elements	Which design model elements have to be changed, e.g., subsystems, components and connectors (if modeled explicitly)?
Execution tasks	How can the AR be applied? Execution tasks may include i) tasks to realize structural changes in a design, dealing with components and connectors as well as subsystems and their interfaces, ii) implementation and configuration tasks in development and operations, and iii) documentation and communication tasks such as modeling activities, technical-writing assignments, or design workshop preparation and facilitation.

REFERENCES

[1] W.H. An, H. Cai, L. Fan, B. Gao, C.J. Guo, L.L. Ma, Z.H. Wang, M.J. Zhou, Locating Isolation Points in an Application Under Multi-Tenant Environment, U.P.T. Office. US, International Business Machines Corporation, 2012.

[2] P. Avgeriou, J. Grundy, J.G. Hall, P. Lago, I. Mistrík, Relating Software Requirements and Architectures, Springer-Verlag, 2011.

[3] L. Beighley, M. Morrison, Head First PHP & MySQL, O'Reilly Media, Sebastopol, 2008.

[4] C. Bisig, Ein werkzeugunterstütztes Knowledge Repository für Architectural Refactoring, Masters thesis, HSR Hochschule für Technik Rapperswil, 2016.

[5] J. Boeder, B. Groene, Architecture of SAP ERP: Understand How Successful Software Works, tredition, 2014.

[6] G. Chang Jie, S. Wei, H. Ying, W. Zhi Hu, G. Bo, A framework for native multi-tenancy application development and management, in: The 9th IEEE International Conference on E-Commerce Technology and the 4th IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services, 2007, CEC/EEE 2007, 2007.

[7] J. Coggeshall, Zend Enterprise PHP Patterns, Springer, Dordrecht, 2009.

[8] Cor-Paul Bezemer, A. Zaidman, Challenges of Reengineering into Multi-Tenant SaaS Applications, Delft University of Technology, 2010.

[9] M. Cross, Developer's Guide to Web Application Security, Syngress Press, San Diego, 2007 [Imprint].

[10] D. Duggan, Enterprise Software Architecture and Design: Entities, Services, and Resources, Wiley, Hoboken, 2012.

[11] C. Fehling, F. Leymann, R. Retter, W. Schupeck, P. Arbitter, Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications, Springer, Dordrecht, 2014.

[12] M. Fowler, Patterns of Enterprise Application Architecture, Addison-Wesley, Boston, 2003.
[13] Frederick Chong, Gianpaolo Carraro, R. Wolter, Multi-tenant data architecture. Retrieved from http://msdn.microsoft.com/en-us/library/aa479086.aspx, 2006 (accessed 18 May 2016).
[14] E. Freeman, E. Robson, B. Bates, K. Sierra, Head First Design Patterns, O'Reilly Media, Sebastopol, 2008.
[15] A. Goucher, T. Riley, Beautiful Testing: Leading Professionals Reveal How They Improve Software, O'Reilly Media, Sebastopol, 2009.
[16] R. Hanmer, Patterns for Fault Tolerant Software, Wiley, 2013.
[17] G. Hohpe, B. Woolf, Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions, Addison-Wesley, Boston, London, 2004.
[18] K. Julisch, C. Suter, T. Woitalla, O. Zimmermann, Compliance by design – bridging the chasm between auditors and IT architects, Comput. Secur. 30 (6–7) (2011) 410–426.
[19] J. Kabbedijk, C.-P. Bezemer, S. Jansen, A. Zaidman, Defining multi-tenancy: a systematic mapping study on the academic and the industrial perspective, J. Syst. Softw. 100 (2014) 139–148.
[20] J. Kerievsky, Refactoring to Patterns, Addison-Wesley, Boston, 2005.
[21] G. Kousiouris, D. Kyriazis, A. Menychtas, T. Varvarigou, Legacy applications on the cloud: challenges and enablers focusing on application performance analysis and providers characteristics, in: 2012 IEEE 2nd International Conference on Cloud Computing and Intelligent Systems (CCIS), 2012.
[22] M. Kun, C. Zhenxiang, A. Abraham, Y. Bo, S. Runyuan, A transparent data middleware in support of multi-tenancy, in: 2011 7th International Conference on Next Generation Web Services Practices (NWeSP), 2011.
[23] K. Mcarthur, Pro PHP, Springer, Dordrecht, 2008.
[24] P. Mell, T. Grance, The NIST definition of cloud computing, http://dx.doi.org/10.6028/NIST.SP.800-145, 2011.
[25] G.J. Myers, T. Badgett, T.M. Thomas, C. Sandler, The Art of Software Testing, John Wiley & Sons, Inc., Hoboken, NJ, 2004.
[26] S. Newman, Building Microservices, O'Reilly Media, Inc., 2015.
[27] M. Pistoia, S. Chandra, S.J. Fink, E. Yahav, A survey of static analysis methods for identifying security vulnerabilities in software systems, IBM Syst. J. 46 (2) (2007) 265–288.
[28] C. Ramakrishnan, Data leakage detection in a multi-tenant data architecture, M. Corporation, United States Patent Application US 2014/0130175 A1, 2014.
[29] A. Saray, Professional PHP Design Patterns, Wiley, Hoboken, 2009.
[30] B. Sengupta, A. Roychoudhury, Engineering multi-tenant software-as-a-service systems, in: Proceedings of the 3rd International Workshop on Principles of Engineering Service-Oriented Systems, ACM, Waikiki, Honolulu, HI, USA, 2011, pp. 15–21.
[31] G. Shroff, Enterprise Cloud Computing: Technology, Architecture, Applications, Cambridge University Press, Cambridge, 2010.
[32] Y. Singh, Software Testing, Cambridge University Press, Cambridge, 2011.
[33] D. Spinellis, G. Gousios, Beautiful Architecture: Leading Thinkers Reveal the Hidden Beauty in Software Design, O'Reilly Media, Sebastopol, 2009.
[34] F. Trucchia, J. Romei, Pro PHP Refactoring, Apress, 2010.
[35] M.A. van der Linden, Testing Code Security, Taylor & Francis, Hoboken, 2007.
[36] A. Vance, The cloud: battle of the tech titans, Bloomberg Business. Retrieved from http://www.bloomberg.com/bw/magazine/content/11_11/b4219052599182.htm, 2011, p3 (accessed 18 May 2016).
[37] w3techs, Usage statistics and market share of PHP for websites. Retrieved from https://w3techs.com/technologies/details/pl-php/all/all, 2014 (accessed March 2017).
[38] J. Weinman, Cloudonomics: The Business Value of Cloud Computing, Wiley, Hoboken, 2012.
[39] Z. Xuesong, S. Beijun, T. Xucheng, C. Wei, From isolated tenancy hosted application to multi-tenancy: toward a systematic migration method for web application, in: 2010 IEEE International Conference on Software Engineering and Service Sciences (ICSESS), 2010.
[40] M. Yener, A. Theedom, Professional Java EE Design Patterns, Wiley, Hoboken, 2014.
[41] M. Zandstra, PHP Objects, Patterns, and Practice, Springer, Dordrecht, 2010.
[42] Zend Technologies Ltd., Programmer's reference guide of Zend framework 2. Retrieved from http://framework.zend.com/manual/2.3/en/index.html (accessed 18 May 2016).
[43] O. Zimmermann, Architectural refactoring: a task-centric view on software evolution, IEEE Softw. 32 (2) (2015) 26–29.

[44] O. Zimmermann, Architectural refactoring for the cloud: a decision-centric view on cloud migration, Computing (2016), http://dx.doi.org/10.1007/s00607-016-0520-y. Retrieved from http://link.springer.com/article/10.1007/s00607-016-0520-y (accessed 24 October 2016).

[45] O. Zimmermann, L. Wegmann, H. Koziolek, T. Goldschmidt, Architectural decision guidance across projects-problem space modeling, decision backlog management and cloud computing knowledge, in: 2015 12th Working IEEE/IFIP Conference on Software Architecture (WICSA), IEEE, 2015.