Customization Realization in Multi-tenant Web Applications: Case Studies from the Library Sector

Slinger Jansen[1], Geert-Jan Houben[2], and Sjaak Brinkkemper[1]

[1] Utrecht University, P.O. Box 80.089, 3508TB Utrecht, The Netherlands
[2] Delft University of Technology, P.O. Box 5031, 2600 GA Delft, The Netherlands

Abstract. There are insufficient examples available of how customization is realized in multi-tenant web applications, whereas developers are looking for examples and patterns as inspiration for their own web applications. This paper presents an overview of how variability realization techniques from the software product line world can be applied to realize customization when building multi-tenant web applications. The paper addresses this issue by providing a catalogue of customization realization techniques, which are illustrated using occurrences of customization in two practical innovative cases from the library sector. The catalogue and its examples assist developers in evaluating and selecting customization realization techniques for their multi-tenant web application.

1 Introduction

Web applications profit greatly from customization, as they make them applicable in a broader context, enable component reusability, and make them more user-specific. Especially when taking the software as a service model into account, the benefits of one centralized software application with multiple tenants, over the alternative of multiple deployments that have to be separately maintained, become apparent. Customization and multi-tenancy, however, do not come for free. In customized web applications code becomes more complex, performance problems impact all tenants of the system, and security and robust design become much more important. A multi-tenant web application is an application that enables different customer organizations ('tenants') to use the same instantiation of a system, without necessarily sharing data or functionality with other tenants. These tenants have one or more users who use the web application to further the tenant's goals. Software developers implementing these systems have all at some point wondered what the available Customization Realization Techniques (CRTs) are. This leads to the following research question:
How are customization and configurability realized in Multi-tenant web applications?

There are three research areas that are directly related to CRTs in multi-tenant web applications: variability in software product lines, end-user personalization in web applications, and multi-tenancy architectures.

From the area of software product lines variability models can assist in determining techniques for customization as is already determined by Mietzner [1]. Svahnberg, van Gurp, and Bosch [2] present a taxonomy of variability realization techniques, some of which were previously encountered in a case study [3]. These variability realization techniques, however, are only partially relevant for two reasons: (1) web applications are generally one-instance systems and (2) the variability realization techniques

B. Benatallah et al. (Eds.): ICWE 2010, LNCS 6189, pp. 445–459, 2010.

of Svahnberg have different binding times, whereas variability in web applications is generally bound at run-time only.

Rossi et al. [4] introduce the following scenarios for web application customization: static customization, link customization, node structure customization, node content customization, and context customization. These scenarios are interesting, but do not include customizations that impact the structure of the web application, such as the integration with another (varying) application or the adaptation of the object model on which the system is built. Chong and Carraro [5] suggest there are four ways to customize web applications in a multi-tenant environment, being user interface and branding, workflow and business rules, data model extensions, and access control. The practical view of Chong does include data model extensions, but also does not take into account integration with varying other applications. Another specific problem that is not covered by the customizations of Chong is user interface adaptations based on user specific properties, such as experience level.

WUML [6] is a modeling language that enables customization by splitting up functionality in a generic and a customized part. Unfortunately, this technique only enables a priori customization and end-user customization is not possible at present. A generic framework for web site customization is introduced by Lei, Motta and Domingue [7], that provides customization capabilities based on any logic condition for web-based applications. The framework is based on the mature OntoWeaver web application modeling and development framework. The capabilities of the customization framework appear to be geared towards the views and controller of applications. The framework does not cover, however, customization by linking to other applications, accessing different databases with one code base, or customer-specific model adjustments.

Mietzner et al. [1,8] propose a variability modeling technique for multi-tenant systems. The variability modeling technique is also rooted in the software product line research area, and defines internal (developer view) and external (customer view) variability. Their variability modeling technique enables developers to speculate and calculate costs for unique instances within a multi-tenant deployment. Furthermore, Mietzner also addresses different deployment scenarios for different degrees of tenant variability. Unfortunately, they do not address the CRTs. Guo et al. [9] provide a framework for multi-tenant systems. Their framework makes a relevant distinction with regards to security, performance, availability, and administration isolation, where issues are addressed that would not occur in a single-tenancy environment. The framework also aims to provide as much flexibility for end-users, even to the extent that end-users create their own customizations on the fly. Again, the issue of how such customizations are implemented is not addressed.

In regards to customization and its relationship to end-user personalization, much can be learned about the reasons for changing web application functionality based on a user's context. Goy et al. [10] state that customization is based on information about the user (knowledge, interests and preferences, needs, and goals), the device used by the user to interact with the system, and information about the context of use (physical context, such as location, and social context). Schilit, Adams, and Want define context as the 'where you are', 'who you are with', and 'what resources are nearby' [11]. Such information is generally stored in a user model [12,13]. Ardissono et al. [13]

address the problem of web application customization using an advanced evaluation model that questions the user on his interests and that automatically derives what parts of the application the user is interested in. Fiala and Houben present a tool that adapts a web page after it has been generated but before it is presented to the user, using a set of transcoding rules [14]. The models used for these techniques are, however, not re-usable for multi-tenant systems, since the meta-configuration data stored per tenant is dissimilar from the information that is stored per user in these user models. It must be noted that most literature that refers to such user models [12,11] addresses dynamic adaptation of web applications, generally to improve the system behaviour for specific end-users, such as showing or removing options based on the level of the end-user [12]. The dynamic adaptations, however, conflict directly with the topic of this paper, i.e., the description of static customizations per tenant. The work on these user models and customization is therefore considered out of scope.

In this paper we present a catalogue of CRTs for web applications and show how these are implemented in practice using two case studies. The catalogue assists web application developers and architects in selecting the right techniques when implementing multi-tenant web applications. Without such a catalogue, developers of multi-tenant web applications need to reinvent the wheel and are unable to profit from existing architectural patterns used for implementing multi-tenancy.

This research was initiated because a lack was noticed in current literature regarding customization techniques in web applications, specifically with regards to multi-tenant web applications. The aim was set to create an overview of such techniques for developers, such that they can reuse these techniques and their associated implementation details. The research consisted of three steps. First, the two case studies were performed, using the theory-building multi-case study approach [15,16]. The case study results were discussed with and confirmed by lead developers from both cases. Secondly, the CRTs overview was created. Thirdly, this overview was evaluated by five developers of multi-tenant web applications, who were not part of the core development teams of the case studies. In both cases the first author played a major part as lead designer. The developers who were interviewed were all at some point involved in the development of multi-tenant systems. They were interviewed and in-depth discussions were had about the proposed CRTs.

Section 2 continues with a discussion of the different types of customization. Sections 3 and 4 describe the two case studies of native multi-tenant library web applications and the customization techniques encountered. Finally, in sections 5 and 6 the findings and conclusions are presented.

2 Definition of Core Concepts: Multi-tenancy and Customization

In a multi-tenant web application there are tenants (customer companies) and the tenant's users. Users are provided with features that may or may not be customized. A feature is defined as a distinguishing characteristic of a software item (e.g., performance, portability, or functionality). A customized feature is a feature that has been tailored to fit the needs of a specific user, based on the tenant's properties, the user's context and properties, or both. Customized features are customized using customization

techniques, which are in turn implemented by CRTs. These are implemented by variability realization techniques.

Variability realization techniques are defined by Svahnberg, van Gurp, and Bosch [2] as the way in which one can implement a variation point, which is a particular place in a software system where choices are made as to which variant to use. Svahberg's variability realization techniques have specific binding times, i.e., times in the software lifecycle at which the variability must be fixed. Because the systems under study are multi-tenant systems, it is impossible to apply all variability realization techniques into account that have a binding time that is not at run-time, since there is just one running instance of the system. This constraints to the following applicable variability realization techniques: (a) infrastructure-centered architecture, (b) run-time variant component specializations, (c) variant component implementations, (d) condition on a variable, and (e) code fragment super-imposition. Typically, in a multi-tenant environment a decisions must be made whether to store a tenant data in a separate databases or in one functionally divided database [5]. By following the variability techniques of Svahnberg this major decision is classified as 'condition on a variable'. The same is true, however, if the tenant requires a change in the view by adding the tenant's logo to the application. The difference between these two examples is the scale on which the change affects the architecture of the web application. Svahnberg's techniques provide some insight into the customization of web applications, but a richer vocabulary is required to further specify how customizations are implemented in practice. The richer vocabulary is required to identify patterns in the creative solutions that are employed to solve different functional customization problems. To bring variation realization techniques and customized features (subject to developer creativity) closer together, the CRTs are introduced.

Before we continue providing the CRTs, the descriptions of the variability realization techniques of Svahnberg et al. are repeated here in short. The (a) infrastructure-centered architecture promotes component connectors to a first class entity, by which, amongst other things, it enables to replace components on-the-fly. (b) Run-time variant component specializations enable different behaviors in the same component for different tenants. (c) Variant component implementations support several concurrent and coexisting implementations of one architectural component, to enable for instance several different database connectors. (d) Condition on a variable is a more fine-grained version of a variant component specializations, where the variant is not large enough to be a class in its own right. Finally, (e) code fragment super-imposition introduces new considerations into a system without directly affecting the source code, for instance by using an aspect weaver or an event catching mechanism.

Five CRTs can be identified from two types of customization: *Model View Controller (MVC) customization* and *system customization*. By MVC customization we mean any customization that depends on pre-defined behaviour in the system itself, such as showing a tenant-specific logo. We base it on the prevalent model-view-controller architectural pattern [17], that is applied in most web applications as the fundamental architectural pattern. In regards to the category of MVC customization we distinguish three changes: *model changes*, *view changes*, and *controller changes*. By system customization we mean any customization that depends on other systems, such as a

tenant-specific database or another web application. For the category of system customization two types of changes are distinguished: *connector changes* and *component changes*. Please note that each of the changes can be further specialized, but presently we aim to at least be able to categorize any CRT.

The first customization is that of **model change**, where the underlying (data-)model is adjusted to fit a tenant's needs. Typical changes vary from the addition of an attribute to an object, to the complete addition of entities and relationships to an existing model. Depending on the degree of flexibility, the ability to make model changes can be introduced at architecture design time or detailed design time. Binding times can be anywhere between compilation and run-time, again depending on the degree of freedom for the tenants and end-users. An example of an industrial application is SalesForce[1], in which one can add entities to models, such as domain specific entities. The variability for SalesForce was introduced at architecture design time and can be bound at run-time. The model change assumes there is still a shared models between tenants. If that is not the case, i.e., tenants get a fully customized model, the customization is considered a full component change (being the model itself).

The second type of customization is that of **view change**, where the view is changed on a per-tenant basis. Typical changes vary from a tenant-specific look and feel, to complete varying interfaces and templates. Again, depending on the degree of flexibility, variations can be introduced between design time and run-time. Binding times can be anywhere between compilation and runtime, also depending on the degree of freedom for the end-users. An example of an industrial application is the content management system Wordpress, in which different templates can be created at runtime to show tenant-specific views of the content.

Controller change is the third type of customization, where the controller responds differently for different tenants and guides them, based on the same code, through the application in different ways. The simplest example is that of tenant specific data, which can be viewed by one tenant only. More extreme examples exist, such as different license types, where one tenant pays for and uses advanced features, opposed to a light version with simple features for a non-paying tenant. Binding times are anywhere between design time and runtime and again depend on how much freedom the tenants are given. An example of an industrial application is the online multi-tenant version of Microsoft CRM, which enables the tenant to create specific workflows for end-users.

The system changes are of a different magnitude: they concern the system on an architectural level and use component replacement and interfaces to provide architectural variability. The fourth type of customization is **system connector change**, where an extension that connects to another system is made variable, to enable connecting to different applications that provide similar functionality. An example might be that of two tenants that want to authenticate their users without having them enter their credentials a second time (single-sign-on), based on two different user administration systems. The introduction time for any system connector change is during architecture design time. The binding time can be anywhere after that up to and including during runtime. A practical example is that of photo printing of Google's Picasa, a photo management application. Photo printing can be done by many different companies and depending on

[1] `http://www.force.com`

Table 1. Customization techniques, their realization techniques, and their latest introduction times

Customization Realization Technique	Latest introduction time	a	b	c	d	e
Model change	Design		✓		✓	
View change	Detailed design		✓		✓	✓
Controller change	Detailed design				✓	✓
System connector change	Architecture design		✓	✓	✓	
System component change	Architecture design	✓		✓	✓	

(a) Infrastructure centered architecture, (b) Run-time variant component specialization, (c) Variant component implementations, (d) Condition on variable, (e) Code fragment super-imposition

which company suits the needs of the Picasa user, he or she can decide which company (and thus connector) will be most suitable for his or her picture printing needs.

Finally, the fifth type of customization is **system component change**, where similar feature sets are provided by different components, which are selected based on the tenants' requirements. An example is that of a tenant that already developed a solution for part of the solution provided by a multi-tenant web application that the tenant wants to keep using. The component in the web application is, in this example, replaced by the component of the tenant completely. Depending on the level of complexity, further integration might be needed to have the component communicate with the web application, when necessary. Introduction time for system component changes is during architectural design. Binding time can be anywhere between architectural design and runtime. A practical use of the system component change is that of Facebook, a social networking site, which enables an end-user to install components from both Facebook and third parties to gain more functionality. System component changes also include the provision of a tenant-specific database or a tenant-specific model.

Table 1 lists the customization techniques of our model and their latest time of introduction. These customizations have their latest binding time at run-time, since all tenants use the same running instance of the system. The introduction time, however, is during either architecture design or detailed design of the web application. For the model change technique, for instance, it must be taken into account during architecture design that end-users might be able to add to the object model of a web application.

The CRTs presented here provide deeper insight into how tenant-specific customizations can be created in multi-tenant web applications. The list assists developers in that it provides an overview of ways in which customization can be implemented for multi-tenant systems. These techniques are elaborated and embellished with examples from the two case studies in sections 3 and 4, to further help developers decide what practical problems they will encounter when implementing such customization.

3 Case Study 1: Collaborative Reading for Youngsters

In 2009 a program was launched in a Dutch library that aims to encourage reading in public schools for youngsters between the ages of eight and twelve. The aim within the

Fig. 1. Screenshot Collaborative Reading Support Information System

program is to have groups of students read one specific book over a short period of time in a so-called reading club. In that time, several events are organized at local libraries, museums, and schools, to further encourage the student to actively participate in the reading process. These events consist of reading contests (who reads the most excitingly and beautifully out loud?), interviews with writers, airings of screenings of the book, poetry contests, and other related activities. The children involved are informed using a custom built content management system. A screenshot is shown in figure 1. The program has been running for approximately one year. The program was launched by one library, which means that other libraries and schools pay the initiating library a fee per participant. Projects can run simultaneously, but also on different moments in time. Tenants are libraries wishing to run a series of reading projects. End-users are members of the library in the age group of 10-12 years old.

3.1 Tenant-Specific Customization

No **model** or **view changes** were made for the Reading Support Information System (RSIS). All end-user and tenant-specific customizations are realized in the RSIS using the **controller change technique**. The controller customizations are complex, however, due to the unique views on the content for each user. For example, the RSIS provides children with a child-specific view of the project in which they participate. The view includes an activity agenda, a content management system on which both participants and organizers can post, a shoutbox (a place where short unfiltered messages can be posted by members on the frontpage), a reading progress indicator, and a project information page. The content management system enables organizers and participants to post polls, pictures, movies, and basic textual blog items. The RSIS also has some basic

social network facilities, which enables users to befriend each other and enables users to communicate. The system is not fully localized to one tenant. Participants from two different tenants can see each other's profiles, view each other's blog posts, and become friends using the social networking facilities, to enable two participants from different areas in the country to connect.

Content is also only in part limited to the tenant. Most of the content, such as the local organizer's content items and that of the participants can only be seen (by default) in the local scope. Some of the content, however, should be published nationally, when two projects are running in different locations simultaneously and the content item spans multiple projects. On the other hand, when a local organizer chooses to edit a national item, a new instance of that content item is created that becomes leading within the tenant's scope. The tenant can choose to revert to the previous version, deleting the tenant-specific version.

The changes to the controller are complex and required a lot of testing. During the first pilot projects it was discovered that members could sometimes still see content from other tenants, such as in the shoutbox, where different projects were run. Also, a downside was discovered when doing performance testing: the home page requires several complex queries for every refresh, thereby introducing performance problems. The main upside of multi-tenancy, which is found in having to maintain one deployment across different customers, was almost removed completely by the performance issue. Several solutions were implemented, such as deploying the shoutbox on a dedicated server, and some query optimization. In all encounters of customization, controller changes were implemented using the tenant variable as the configuring condition, i.e., if the end-user belongs to a library, the end-user stays in that particular scope.

No other customizations were necessary: the **model** does not require extensions, the **views** must look uniform for different tenants, there are no **connectors to external components**, and none of the **components** need to be replaced. One could argue that some changes to the view are made since local organizers (libraries) can customize the view offered to participants by adding a tenant specific logo. However, this logo is stored in the database, and the controller determines which logo to show, which by our classification makes it a controller change.

4 Case Study 2: Homework Support System for Schools

In 2007 the proposal for a Homework Support System (HSS) for secondary schools was approved by an innovation fund for public libraries. The idea was to set up a new web service for high school students, where help is provided to perform tasks like show-and-tell speeches, essays and exam works for which bibliographic investigation is required. The platform now forms the linking pin between public libraries, teachers, students, and external bodies such as schoolbook publishers, parents, and homework assistance institutes. The goal of the HSS is to increase the adoption of the public library for homework assignments by the target group of high school students, to provide education institutions with a tool to help them organise their work and save time, and for libraries to remain an attractive place for students to meet and work, both online and on-premise. The role of the public library as a knowledge broker would be, if the homework support

system is successful, reaffirmed in the age group of 12–20. This age group is well-known for its disregard for the public library, whereas on the other hand they experience lots of difficulty when searching on the web with commercial search engines [18]. The tenants are libraries and the users are members of the library in the age group 12–20.

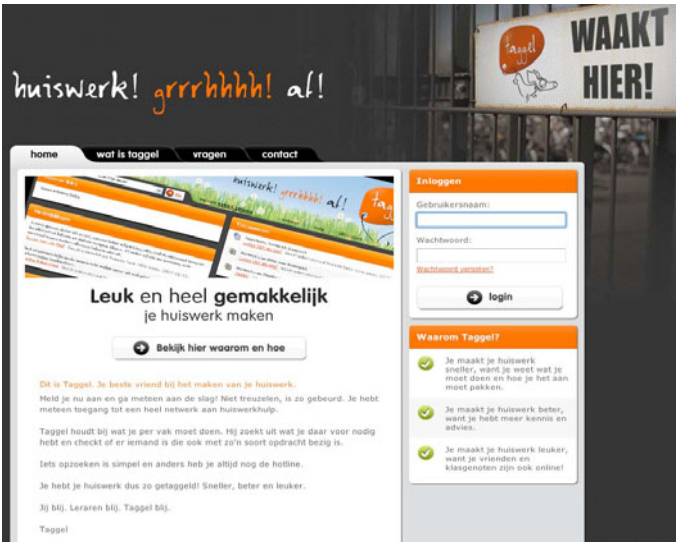

Fig. 2. Screenshot Collaborative Homework Support Information System

The HSS (aptly named Taggel.nl, screenshot in figure 2) that was developed is a web application that helps students find reliable sources of information while doing research for homework projects. The HSS consists of a number of main components being groups (management), a search component, a collaboration component, a knowledge link and rating component, and a portfolio component. Presently, the HSS is running a pilot with three schools and five libraries in one Dutch region. Results have been promising, but many technical challenges are still to be solved.

The HSS consists mostly of commercial and off-the-shelf components. These components are modeled in figure 3. Central to the HSS architecture is the HSS controller, a relatively light-weight navigation component that coordinates each user to the right information. The controller produces views for students and fills these views with data from the database, which is based on the object model of the application. The object model stores students, libraries, and schools as the main entities. When a student logs in through the controller, the associated school and library are determined. Data is synchronized regularly with the Student Administration System (SAS). Different SASs can be used, based on a configuration option that is set by the developer or administrator of the system. The HSS also communicates with the national library administration system. When the system has confirmed that the user is a member of the library, the user gains several extra features, such as automatic book suggestions in the e-learning environment and book reservation.

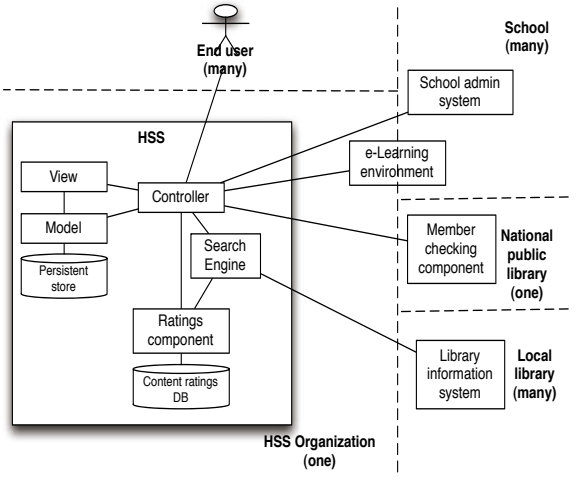

Fig. 3. Component and Component Location Overview of HSS

4.1 Tenant-Specific Customization

With regards to CRTs, the HSS is more interesting than the RSIS, since the HSS employs almost all the possible CRTs. Customizations are triggered by several tenant- and user-specific properties. Each of the different customization types will be discussed here, including a rationale and the advantages and disadvantages of using the CRT.

There are no changes to the **models** or **views**, since no custom model extensions are required (the domain was known at design time) and the system must look uniform across different tenants.

In regards to the **controller changes**, several customizations are specific to a tenant. First, end-users have specific scopes in which they are active, such as their school, classroom, or library. The scope determines what content they can see, in which discussion groups they can take part, and to which library a homework question is directed. The scope is determined by either the library or the school of which the student is a member. This scoping is a typical instrument used in multi-tenancy systems, as was already seen in the RSIS case study, the main advantage being that different tenants hide data from each other. Much like in the RSIS case, scoping was complex compared to traditional multi-tenant web applications, because some content is shared between tenants.

The controller component controls access to Fronter, a commercial e-learning platform. The e-learning platform presently provides a lot of the functionality of the HSS, and its Application Programming Interface (API) is used to develop other applications that interact with the e-learning platform. The platform provides the portfolio component, the collaboration component, and the groups functionalities. Furthermore, it provides a real-time chat functionality, enabling librarians to provide real-time support to students. If the school already uses Fronter or wants to use another e-learning platform, it can replace the multi-tenant version of Fronter used for HSS. Simple html page encapsulation is used to capture Fronter pages in the HSS, though more integration can be and has been implemented using the other CRTs.

In regards to the **interface change**, several CRTs are used for the HSS. First, students have to be able to search in the databases of their local libraries. Secondly, in case the schools already have single sign-on solutions, these solutions can be used to sign on to the HSS, and different component interfaces were built to support single sign-on. The rationale behind both is simple: per tenant different systems need to be approached with different interfaces. The development of the interfaces was relatively simple, since in both cases previous solutions could be reused.

In regards to **component change**, customization is required for schools that already have an e-learning environment to replace the HSS supported environment. The Fronter component is in this case replaced by an already deployed solution on the school's servers. Presently, this tenant specific customization is well supported by the architecture, since it only requires some links to change. However, the true integration between systems in this case is lost. For example, it is presently possible to add any source found in the search engine, such as a book or an online newspaper article, to any content, such as a homework assignment or discussion, in the e-learning environment with one click. When the component is replaced and no new integration is applied, this ceases to work. In the development of the HSS was found that component change introduces complexity on an organizational level, where suddenly schools are confronted with adjustments to their own systems. To enable the schools in solving these problems, an API has been constructed, which enables schools with the same student information system products to exchange and independently maintain proprietary SAS API code.

At present, the system interface changes are built into the HSS rather haphazardly. Customization code is built into HSS (catching specific events) in different places. Mixing system customization and tenant customization code is generally considered bad practice, and will soon be separated into several separate code files.

4.2 Combining Mechanisms for Advanced Search

An example of the combination of two CRTs is found in the search engine. The HSS was designed to provide results ordered by quality and by relevance for a student, depending on the student's context (student level, school programme, source quality, locality). Furthermore, the HSS was designed to obtain these search results from the HSS content database and from a library specific database that contains all books and materials that can be taken out of that local library. It has not been visualized in figure 3, but there are in fact two sources that feed the search results, which are in turn ordered by yet another source with content ratings from librarians and school teachers.

The customization of these search results is taken care of by the search component, which can be considered part of the controller component of the HSS. customization happens on three levels, being (1) which library database to pick, (2) which content to show, and (3) in which order to show that content. The library database is chosen based on the library of which this student is a member, or, if that library does not participate in the HSS project, the library that is closest to the school that participates (it is not uncommon for Dutch schools to have a steady relationship with the nearest local library). The content that is shown from the HSS database is based on the school the student goes to, although most content in the local HSS database presently is public.

The customization mechanisms used are system connector change and controller change. The controller (i.e., search component) does most of the work here, since it has the responsibility to determine how queries are sent to which library catalogue instance (system connector change), how search results are ordered (controller change), and which results are shown (controller change). The variability mechanisms used are conditional variables that are stored in the database, i.e., not all variants were made explicit when the system was started up. Please note that the order is determined by the ratings component, which contains content quality and level ratings from teachers and librarians.

5 Case Study Findings

Table 2 lists the encountered CRTs in the two cases studies. The table also shows which variability realization technique has been employed for realizing the CRT. The 'controller change' CRT is encountered most frequently. This is not surprising, since it is easy to implement (compared to for instance model changes), has little architectural overhead, can be used when the number of variations is implicit or explicit, and the mechanism can be reused over the complete web application. Also, in both cases the model was pre-defined and no further changes were needed to the views (on the contrary, both systems have to present a uniform look and feel). One thing must be noted about controller changes, and that is that controller change code must be tested extensively. This is even more so when the controller shares some content across different tenants, which can introduce privacy problems when one tenant erroneously sees sensitive data from another tenant.

System customizations introduce challenging architectural challenges. In regards to architecture, depending on the direction of the data flow (inbound or outbound to the system) different architectural patterns can be used to arrange communication between the systems. An example of such an architectural pattern is the use of the API of the other system, such as in the case of the single sign-on feature of the HSS, through varying component implementations. The use of such an API requires the developers to know the number of variations a priori. Another example is that of an infrastructure centered architecture that only starts to look for available components at runtime, such as the example that the e-learning component can be replaced dynamically, based on one variable. Since the e-learning component runs in a frame of the HSS, it can easily be replaced. Of course, integration with other functionality of the e-learning system has to be built in again.

Even larger *organizational* challenges are introduced when doing system customizations. Whereas the developers and architects generally have a 'it can be built' attitude, customer organizations are reluctant to add new functionality to their existing systems and these co-development projects (both the school and HSS team, for instance) take far more time than necessary when looking at the actual code that is built in. For example, the e-learning environments of schools are generally hard to adjust, if possible at all. Also, because the interface code that is built needs to be maintained, this introduces considerable increase in total cost of ownership for the tenants. The introduction of generic

Table 2. Customization Realization Techniques used in the Two Cases

Case	Functionality	Customization parameters	Customization realization	Variability realization technique
1	Social networking	Tenant	Controller change	(d)
1	Content views	Tenant	Controller change	(d)
1	Content add	Tenant	Controller change	(d)
1	Calendar	Tenant	Controller change	(d)
1	Shoutbox	Tenant	Controller change	(d)
1	Projects	Tenant, payment, date	Controller change	(d)
2	Discuss with peers	School	Controller change	(d)
2	Access school content	School	Controller change	(b)
2	Search in book and KB	Library, Student level	Controller change	(d)
2	Discuss with librarian	Library	Controller change	(d)
2	Be active in group	Student context	Controller change	(d)
2	Access library database	Library	System interface ch.	(b), (d)
2	Discuss with teacher	School, SAS	System interface ch.	(c), (d)
2	Access SAS for single sign-on	School	System interface ch.	(c), (d)
2	Reuse functionality in e-learning environment	School	System component change	(a), (d)

(a) Infrastructure centered architecture, (b) Run-time variant component specialization, (c) Variant component implementations, (d) Condition on variable, (e) Code fragment super-imposition. SAS = School Administration System, KB = Knowledge Base

APIs could help here, but in the case of e-learning environments such generic APIs will not be available in the near future. These APIs can be seen as enablers for an implicit number of variations using system interface change, whereas without a generic API one can only have a limited number of components.

Another finding from these cases is that the binding time for each of the customizations determines how much effort needs to be spent on the customization. Generally, the later the binding time, the larger the effort to build the multi-tenant customization in the system. For example, in both cases the model was known a priori and no adjustments had to be made at run-time by end-users. If such functionality had to be built in, both systems would need an end-user interface and a technical solution that enables run-time model changes. In the two cases these were not required and developers could simply make changes to the models at design time. The same holds for the whether there were an implicit or explicit number of variations. When the number of variations is explicit, such as in the case of the single sign-on solution, the variation points are easy to build in. However, when the number of variations is implicit, such as the number of projects that can be run in the RSIS, the code becomes much more complex because it needs to deal with projects running at differing times that share content and also differing projects running at differing times that do not share content.

The evaluation with developers provided relevant observations, which are summarized as follows. First, they indicated that the customization overview works well on a high level when selecting techniques, but when implementing a technique, several combinations of customization and variability realization techniques will be required.

Also, the second case study was mentioned specifically as being a much better example than the first, since the first case study merely provides insight into some of the simplest mechanisms proposed in the framework. The discussions with developers during the evaluation led to changes to the model: (a) an extra variability realization technique ("code fragment imposition") was added for the CRT "controller change" when one of the developers mentioned that he had implemented such a mechanism. The main advantage of using code fragment imposition, using an event catching mechanism, was that the API would function similar to the web application itself. The developers knew of several examples where the 'wrong' variability realization technique had been used to implement a customization technique. The question was posed whether advice can be given whether a certain type of customization is best applied to a specific situation. The developers listed several negative effects of using the wrong customization technique, such as untraceable customization code over several components, unclear behavior of an application when using code fragment super-imposition, and database switches that were spread out over the whole codebase.

6 Conclusions

It is concluded that current variability realization techniques insufficiently describe multi-tenant customization. This paper provides an overview of CRTs in multi-tenant web applications. These CRTs are further elaborated using examples from two innovative web applications from the library sector. The findings assist developers in choosing their own CRTs and helps them avoid problems and complexities found in the two cases. The provided list of customizations is just the beginning: we consider annotating the CRTs with customization experiences as future work. Typical examples of this knowledge are technical implementation details, code fragments, architectural patterns used, and non-functional requirements implications, such as performance and reliability problems.

References

1. Mietzner, R., Metzger, A., Leymann, F., Pohl, K.: Variability modeling to support customization and deployment of multi-tenant-aware software as a service applications. In: ICSE Workshop on Principles of Engineering Service Oriented Systems, pp. 18–25. IEEE Computer Society, Los Alamitos (2009)
2. Svahnberg, M., van Gurp, J., Bosch, J.: A taxonomy of variability realization techniques: Research articles. Software Practice and Experience 35(8), 705–754 (2005)
3. Jaring, M., Bosch, J.: Representing variability in software product lines: A case study. In: Chastek, G.J. (ed.) SPLC 2002. LNCS, vol. 2379, pp. 15–36. Springer, Heidelberg (2002)
4. Rossi, G., Schwabe, D., Guimaraes, R.: Designing personalized web applications. In: WWW '01: Proceedings of the 10th international conference on World Wide Web, pp. 275–284. ACM, New York (2001)
5. Chong, F., Carraro, G.: Architecture strategies for catching the long tail, Microsoft white paper (2006), `http://msdn.microsoft.com/en-us/architecture/aa479069.aspx`

6. Kappel, G., Pröll, B., Retschitzegger, W., Schwinger, W.: Modelling ubiquitous web applications - the wuml approach. In: Arisawa, H., Kambayashi, Y., Kumar, V., Mayr, H.C., Hunt, I. (eds.) ER Workshops 2001. LNCS, vol. 2465, pp. 183–197. Springer, Heidelberg (2002)
7. Lei, Y., Motta, E., Domingue, J.: Design of customized web applications with ontoweaver. In: K-CAP '03: Proceedings of the 2nd international conference on Knowledge capture, pp. 54–61. ACM, New York (2003)
8. Mietzner, R., Unger, T., Titze, R., Leymann, F.: Combining different multi-tenancy patterns in service-oriented applications. In: Enterprise Distributed Object Computing Conference, IEEE International, pp. 131–140. IEEE Computer Society Press, Los Alamitos (2009)
9. Guo, C.J., Sun, W., Huang, Y., Wang, Z.H., Gao, B.: A framework for native multi-tenancy application development and management. In: E-Commerce Technology and the 4th IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services, CEC/EEE 2007, pp. 551–558 (2007)
10. Goy, A., Ardissono, L., Petrone, G.: Personalization in e-commerce applications. In: Brusilovsky, P., Kobsa, A., Nejdl, W. (eds.) Adaptive Web 2007. LNCS, vol. 4321, pp. 485–520. Springer, Heidelberg (2007)
11. Schilit, B., Adams, N., Want, R.: Context-aware computing applications. In: Proceedings of the Workshop on Mobile Computing Systems and Applications, pp. 85–90. IEEE Computer Society, Los Alamitos (1994)
12. Ardissono, L., Goy, A.: Tailoring the interaction with users in web stores. User Modeling and User-Adapted Interaction 10(4), 251–303 (2000)
13. Ardissono, L., Felfernig, A., Friedrich, G., Goy, A., Jannach, D., Petrone, G., Schäfer, R., Zanker, M.: A framework for the development of personalized, distributed web-based configuration systems. AI Mag. 24(3), 93–108 (2003)
14. Fiala, Z., Houben, G.J.: A generic transcoding tool for making web applications adaptive. In: CAiSE Short Paper Proceedings (2005)
15. Jansen, S., Brinkkemper, S.: Applied Multi-Case Research in a Mixed-Method Research Project: Customer Configuration Updating Improvement. In: Steel, A.C., Hakim, L.A. (eds.) Information Systems Research Methods, Epistemology and Applications (2008)
16. Yin, R.K.: Case Study Research - Design and Methods, 3rd edn. SAGE Publications, Thousand Oaks (2003)
17. Reenskaug, T.: Models, views, controllers, Xerox PARC technical note (December 1979)
18. Fidel, R., Davies, R.K., Douglas, M.H., Holder, J., Hopkins, C.J., Kushner, E.: A visit to the information mall: Web searching behavior of high school students. Journal of the American Society for Information Science (1), 24–37 (1999)