Association for Information Systems
AIS Electronic Library (AISeL)

AMCIS 2009 Proceedings

Americas Conference on Information Systems (AMCIS)

2009

MapperMania: A Framework for Native Multi-Tenancy Business Object Mapping to a Persistent Data Source

Marco Helmich
Hasso Plattner Institute, helmich@hpu.uni-potsdam.de

Juergen Mueller
Hasso Plattner Institute, juergen.mueller@hpi.uni-potsdam.de

Jens Krueger
Hasso Plattner Institute, jens.krueger@hpi.uni-potsdam.de

Alexander Zeier
Hasso Plattner Institute, alexander.zeier@hpi.uni-potsdam.de

Sebastian Enderlein
Hasso Plattner Institute, sebastian.enderlein@hpi.uni-potsdam.de

See next page for additional authors

Authors
Marco Helmich, Juergen Mueller, Jens Krueger, Alexander Zeier, Sebastian Enderlein, and Hasso Plattner

MapperMania: A Framework for Native Multi-Tenancy Business Object Mapping to a Persistent Data Store

Marco Helmich
Hasso Plattner Institute
Marco.Helmich@hpi.uni-potsdam.de

Sebastian Enderlein
Hasso Plattner Institute
Sebastian.Enderlein@hpi.uni-potsdam.de

Jürgen Müller
Hasso Plattner Institute
Juergen.Mueller@hpi.uni-potsdam.de

Jens Krüger
Hasso Plattner Institute
Jens.Krueger@hpi.uni-potsdam.de

Alexander Zeier
Hasso Plattner Institute
Alexander.Zeier@hpi.uni-potsdam.de

Hasso Plattner
Hasso Plattner Institute
Hasso.Plattner@hpi.uni-potsdam.de

ABSTRACT

The Software-as-a-Service delivery model bears new challenges for application developers. Especially in the context of enterprise resource planning software targeting the SME market, new problems arise. Most of them agglomerate around the occurrence of multi-tenancy. This paper describes the framework MapperMania which aims to establish an abstraction layer between the persistence layer and the domain model. Leveraging MapperMania, the domain model is able to abstract from multi-tenancy and changes in the underlying infrastructure.

Keywords

multi-tenancy, data mapping, software as a service, enterprise resource planning, flexibility, customization, business objects.

INTRODUCTION

Enterprise resource planning software mainly occurs in large enterprises. Nevertheless, processes conducted by small and midsize enterprises (SMEs) do have the same complexity (Müller, Krüger, Zeier, 2008). However, SMEs can not afford high customization and maintenance costs. Therefore, they are attracted by the Software-as-a-Service (SaaS) delivery model in which the software is hosted by the software vendor and customers pay recurring subscription fees (Chong and Carraro, 2006).

SaaS completely changes the way enterprise software has to be built in order to address the specific needs of partners and the SaaS provider. On the one hand, customers expect highly flexible enterprise software, custom-tailored to their needs. Regarding the diverse and wide-spread SME market, addressing these customers is rather complicated. Fink and Markovich (2008) recommend an adaptable-horizontal strategy in which an ecosystem of partners develops industry-specific solutions on a rather generic but flexible platform. On the other hand, the SaaS provider aims to leverage economies of scale and anticipates a very homogeneous environment. Consequently following this idea implies the implementation of multi-tenancy (Guo, Sun, Huang, Wang, and Gao, 2007). Multi-tenancy enables the sharing of one application instance among several users and therefore, obviates the need for a dedicated server for each user. This pattern even improves the economies of scale.

In order to succeed in the SME market, a vital ecosystem of partners has to be established. The partners develop industry-specific solutions (so called verticalizations) on this platform and distribute the product for the software vendor (Müller, Krüger, Enderlein, Helmich, Zeier, 2009).

Since enterprise software development today bases on a set of business objects, most partners that develop verticalizations for specific industries take these business objects as a starting point for customization. On the one hand, they obviously need to be customizable and, on the other hand, encapsulate the notion of multi-tenancy. In order to operate the system cost efficiently, business objects need to be shared, which eliminates the possibility for tenant-specific customizations. At first glace, customizing the software while it is shared is impossible.

Therefore, this paper addresses the area of tension between multi-tenancy and customization needs of SMEs. It describes a mechanism how shared business objects can be extended with custom functionality while leaving their source code untouched. Furthermore, it describes a framework that hides the negative effects of multi-tenancy from the developer but incorporates the positive effects. Thus, it describes how an abstraction layer between a multi-tenant persistence and a multi-tenancy-agnostic domain model can be established.

In the course of this paper, a customization project from a company with 300 employees is taken as running example. This company tries to enhance its existing enterprise resource planning (ERP) system with a configurator. In order to achieve that, the business object *Opportunity* needs to be customized. Related to this running example is shown how a business object can be composed tenant-specifically without changing the common code base and persisted in a data store applying the extension table layout (Aulbach, Grust, Jacobs, Kemper, and Rittinger, 2008).

The paper is organized as follows: In the next section, the research conducted is compared to research in related fields and separated against it. The section "Business Object Extension Mechanisms" describes how objects can be extended without changing a shared code base. The section "Data Mapping Framework" describes the implementation of a framework that enables developers to abstract from multi-tenancy leveraging already discussed object extension mechanisms. The last section concludes in final remarks and an outlook is given to further research.

RELATED WORK

Since the emerge of standardized ERP systems, customization became a major concern (Zhang, Lee, Zhang, and Banerjee, 2003; Vilpola, Kouri, and Vaananen-Vainio-Mattila, 2007; Somers and Nelson, 2001; Motwani, Subramanian, and Gopalakrishna, 2005). These considerations become even more important in a market segment that is very diverse and in which every company focuses on its core competencies and, therefore has specialized products (Müller et al., 2008). Standardizing processes in such an environment wipes away potential competitive advantages. Vilpola et al. (2007) developed a framework for decision support which also takes customization features into account. Fink and Markovich (2008) proposed a special strategy to tackle the SME market segment – the adaptable, horizontal strategy. With this strategy the software vendor focuses on providing a rather generic ERP platform and creates business cases for a partner ecosystem. These partners take that business case and build verticalizations for specific industries.

These changes in the business model imply extensive changes in the application architecture (Chong et al., 2006; Chong and Carraro, 2006; Hamilton, 2007).

The chance of success of on-demand business applications was already proven by Salesforce.com. Salesforce.com operates a on-demand, multi-tenant application gaining adequate revenue (Coffee, 2007). In order to achieve a customizable multi-tenant business application, Salesforce.com completely pursues a model-driven architecture in which customer data is separated from metadata (Salesforce, 2008).

(Guo et al., 2007) recommends the introduction of an abstraction layer in order to separate "multi-tenancy aware" developers from "non-multi-tenancy aware" developers. This abstraction layer aims to ease development. Basic guidelines are given in that paper how such an abstraction layer can be implemented.

Compared to multi-tenancy in the business logic and on the user interface, multi-tenancy on database level is well understood. Current research proposes various techniques to improve database performance in a SaaS scenario (Chong et al., 2006; Jacobs and Aulbach, 2007; Aulbach et al., 2008).

However, all these sources are concerned with the implementation of multi-tenancy. Only few considered in academia so far is how multi-tenancy can be encapsulated and how a multi-tenant domain model could look like.

BUSINESS OBJECT EXTENSION MECHANISMS

To overcome the area of tension between multi-tenancy and flexibility, new strategies are required. In order to maximize the positive effects of multi-tenancy (improved cost efficiency), as many resources as possible need to be shared among different tenants. The highest degree of sharing is achieved if the business application (and with that, the business logic) is shared among all tenants.

To reach this degree of sharing while providing a high level of flexibility at the same time, domain objects need to be modified at run-time. On the other side, in the sense of multi-tenancy, changes in the common code base are highly discouraged. This requires a dynamic technology as execution platform. In the following, several concepts enabling object modification at run-time are presented.

Some technologies (e.g. the programming languages Python and Ruby (Müller et al., 2009)) allow to dynamically change the structure of objects at run-time through inheritance. Here a set of structural problems tends to occur - they all are related to the "diamond problem" (Boyen, Lucas, and Steyaert, 1994). The diamond problem is an ambiguity that occurs when a class tries to inherit the same feature over various paths. Other technologies allow weaving additional logic in the application source code.

Mixin-Based Extension

A mixin can be seen as an abstract class providing functionality that can be inherited by a subclass. A mixin is not meant to stand alone. In the actual sense of object orientation, a mixin does not specialize a subclass. Mixins are rather used to collect behavior. Behavior can be collected from multiple mixins via multiple inheritance or extension (Boyen et al., 1994; Bracha and Cook, 1990; Schaerli, Ducasse, Nierstrasz, and Black, 2003). Some technologies even allow this way of extending objects at run-time (see Ruby and Python (Müller et al., 2009)).

With that technique, mixin-based extension opens up a second dimension in object composition. The usual class hierarchy works as the first dimension and allows developers to specify semantic relationships between objects. Completely orthogonal to that works mixin inheritance. Behavior added through mixins does not define a semantic relationship but extends an object with functionality.

Another concept related to mixins are traits (Schaerli et al., 2003). Traits are building blocks of source code similar to mixins but do not leverage the inheritance operation as composition mechanism. The extension of objects is here achieved by another set of operators and runs therefore completely orthogonal to the regular inheritance. Ruby's mixins behave much as these traits since Ruby mixins do not employ the inheritance operation to extend objects and therefore do not change the inheritance graph while extending other objects. Since mixins extend already existing objects, these objects themselves do not need to be changed in order to add functionality.

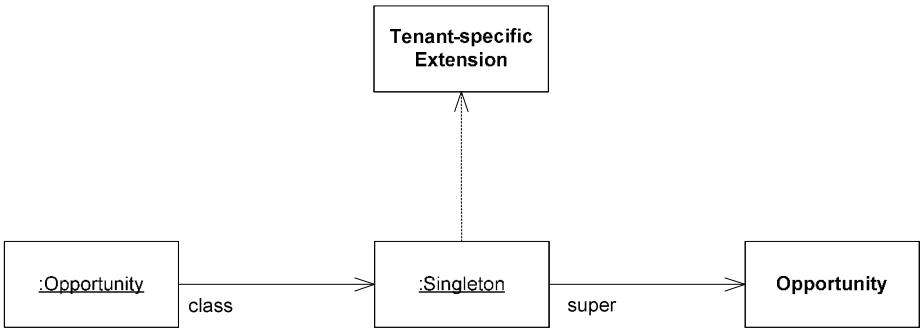

Figure 1. Partner Mixin Extension

On basis of objects that can be modified during run-time of the application, the notion of flexibility is introduced. Tenant-specific source code is placed in mixins. Partners are able to write source code as if they would have a dedicated business object. At run-time, dependent on the tenant who fires the request, the object is composed from these mixins (see Figure 1). This way, additional business logic is bound to specific objects in a tenant-dependent manner. Furthermore, the tenants share all business logic besides these extensions. That means all tenants are able to be operated on the same common code base since all customizations, encapsulated in mixins, are bound dynamically to objects.

DATA MAPPING FRAMEWORK

The solution presented here is called "MapperMania" and is a Ruby library which contains a small domain-specific language (DSL), a mapping engine, and an adapter concept to integrate several data sources. Ruby offers a mixin concept that meets the requirements as described above and is an ideal candidate to implement a DSL (Fowler, 2005; Muller, Fleurey, and Jezequel, 2005).

Domain-Specific Language

Business objects consist of three integral parts: behavior, attributes, and associations to other business objects. Behavior can be mapped to methods. The methods simply can be defined the way Ruby provides. Attributes are mapped to properties, where a *property* is the term used for an attribute that is persistent. Associations to other business objects are mapped to private attributes encapsulated by getter and setter methods. Therefore, the DSL has to define keywords to address all of these business object composites.

Listing 1 shows a simple example of a business object definition. First of all, the DSL syntax, coupled with the data mapping framework MapperMania has to be included in the class. In fact, the syntax of the DSL consists of class methods of the according class or module. Ruby's mixin mechanism is leveraged to bind these class methods to the particular class or module (the methods are available after the *include* statement). These methods now appear to be keywords to the user of this DSL.

```
1   class Opportunity
2     include MapperMania::Resource
3     belongs_to :sales_order
4
5     property :created_at, DateTime, {:nullable => false}
6     property :amount, Fixnum, {:nullable => true, :accessor => :public}
7
8     def increase_amount value
9       self.amount += value
10    end
11  end
```

Listing 1. Business Object Definition

The *belongs_to* keyword defines a one-to-one, navigable association to another business object. These associations are defined on class level which means that each instance of *Opportunity* belongs to an instance of *SalesOrder*. Furthermore, getter and setter methods for this association are introduced. The *belongs_to* statement expects a number of parameters - the first parameter is a symbol specifying the name of the association and the name of the associated class at the same time. Optionally, the associating class can be specified as a second parameter. Moreover, the *belongs_to* statement exists in a second flavor - *has*. The *has* statement defines a one-to-one or one-to-many association to one or more business objects. The multiplicity depends on the first parameter. It can be a Fixnum, a Range, or a not specified number (where "n" is a shortcut). The rest of the syntax is similar to the *belongs_to* statement. The property statement is used to define a persistent attribute of a business object. The first parameter is the name of the property, the second the type. The third parameter is an optional hash containing additional information.

```
1   module OpportunityExtension
2     include MapperMania::Resource
3     extends Item
4     has 1, :configuration
5
6     property :first_name, Fixnum, {:accessor => :private}
7     property :last_name, Fixnum, {:accessor => :private}
8
9     def get_name
10      return self.first_name + self.last_name
11    end
12  end
```

Listing 2. Business Object Extension Definition

This syntax can be used to specify base types of business objects. These are the common business objects that are shared among all tenants. Moreover, the business objects should be customizable without changing shared code. Listing 2 shows an example for business object customization. As already described above, the mixin mechanism is used for customization. Partners that want to modify a business object simply write a module that is mixed in the base type of a certain business object. The base type of an extension is specified with the keyword *extends* that becomes available after the inclusion of MapperMania. It takes, as the only parameter, the class name of the base type it extends. Furthermore, it is possible to define custom methods such as custom getter methods.

Data Mapper

Since the structure of business objects is determined at run-time, MapperMania has to keep information on where and how to store an object. For that purpose, each class and module defines a set of properties and associations. These statements are evaluated when the class is loaded the first time and the used keywords trigger the creation of metadata. This metadata is attached to the according class in the form of property and association sets. These sets contain items that have enough information to construct properties and associations at any given time.

The property and association sets are mainly container classes and provide methods to access particular properties or associations in these sets. Properties store their name and type, visibility information, and whether they are a primary key or not. This information is read by the adapter later on to generate a request to the data store.

Associations store the same information but have one special feature. A priori, associations have two participating entities - one is declared to be the parent and one the child. Since cyclic references may appear and the binding of the instances shall be deferred to run-time, the association is initialized as late as possible. Cyclic references may occur for example when a *SalesOrder* has many *Opportunities* and each *Opportunity* belongs to a *SalesOrder*. This would require that both classes are already loaded. In order to overcome this problem, associations are declared with class names as strings. These names are resolved at instantiation time, when all metadata is present and getter and setter methods for the corresponding object are defined. The binding of associations at run-time enables associations to be declared in modules which means that even a customization may define associations between objects. Therefore associations can be defined tenant-specifically.

This information can now be read from the class in order to generate tables in the database. Since the structure of each instance is supposed to be different, the metadata looks different for each instance. Hence, the metadata has to be specifically composed for each new instance. When a class is instantiated, the instance receives the property and association sets of its base type and all its extensions. This means each instance carries the actual values and the metadata needed to store the values in persistent stores with it.

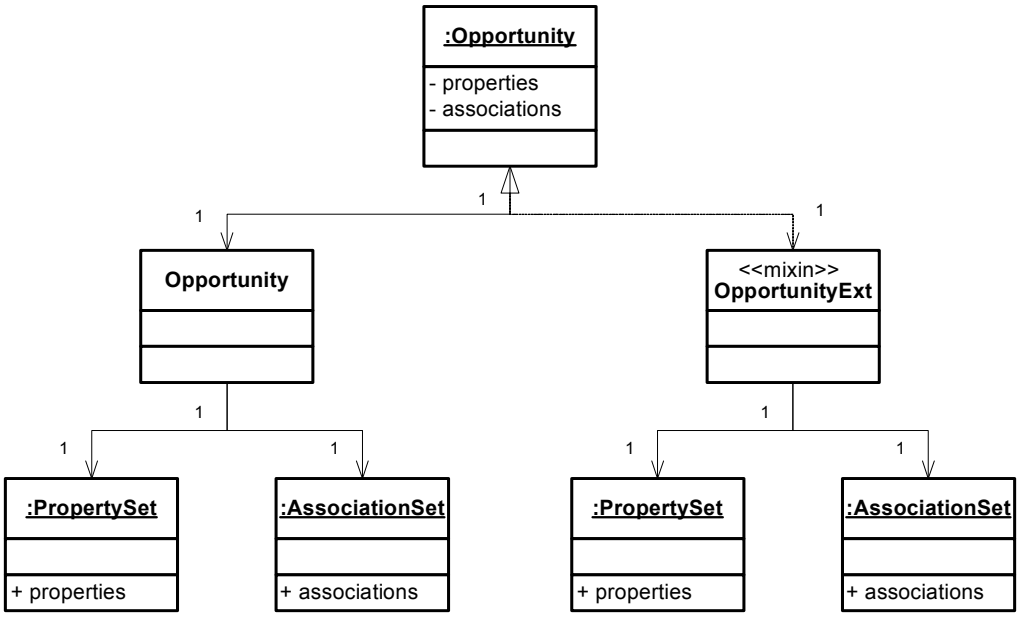

Figure 2. Class Diagram Business Object Instance Example with related PropertySets and AssociationSets

Figure 2 shows a situation in which a business object is extended by a mixin. The instance contains two private variables properties and associations. These variables are arrays containing the property sets and association sets of all classes or mixins they consist of.

In order to guarantee metadata handling for instances, the constructor of each persistent class has to be redefined. Listing 3 depicts extracts of this custom constructor. Moreover, it shows how instances are extended to match their tenant-specific structure. The call in line seven asks the metadata repository to return the extensions of the current tenant. The metadata repository has information on which tenant has which extensions activated and provides MapperMania with this information. Each extension is thereafter classified. After the classification of these extensions, the metadata of these extensions is collected and added to the metadata of the current instance (which is self in Listing 3). Thereafter all metadata is copied, the Ruby extension mechanism is leveraged to bind the methods defined in these extensions to self. At last the getter and setter methods for the custom properties are created. The metadata attached to instances plays an important role during the object mapping. It is the basis of communication to data stores.

```
1   # overrides general constructor
2   # provides constructor method "constructor" to users
3   def initialize
4
5     # extension of current instance with its extensions
6     # this needs to happen before the accessors are built
7     extensions = MapperMania.extensions (self.class.name, MapperMania.tenant_id)
8
9     unless extensions.nil?
10      extensions.each do | ext |
11        klass = self.class.const_get(ext)      # grab class of extension
12        @types << klass                        # merge meta data for ext in self
13
14        ext_props = lass.instance_variable_get("@properties_#{klass.name}".to_sym).deep_clone
15        @properties << ext_props
16
17        ext_assocs=klass.instance_variable_get("@associations_#{klass.name}".to_sym).deep_clone
18        @associations << ext_assocs
19
20        # actually extend self
21        self.extend klass
22      end
23    end
24
25    # stores the properties accessible for the following closure
26    @@array_prop_set = @properties
27
28    # attaching accessors for the properties
29    # has to be made through class_eval due to visibility constraints
30    class << self
31      @@array_prop_set.each do | prop_set |
32        prop_set.properties.each_value do | prop |
33          create_getter prop
34          create_setter prop
35        end
36      end
37    end
38  end
```

Listing 3. Implementation of custom Constructor

Mapping Rule Processor

Mapping rules determine how certain properties are mapped to a data store. Based on the metadata attached to every class and instance, these mapping rules are applied to properties.

In order to concentrate the development effort on the domain logic, the developer shall be able to completely abstract from the notion of a persistence layer. To achieve this abstraction, the inclusion of *MapperMania::Resource* not only introduces keywords necessary for the DSL (see above), but also marks all instances of this class as "able to be persisted". Furthermore, instance methods such as find, save, and destroy become available. These methods are the only ways to control the persistence characteristics of instances.

Listing 4 shows the method responsible for creating an object in a persistent store. Moreover, it is an example how the implementation of a mapping rule can look like. Based on the chosen extension table mapping schema and the metadata (the metadata of the current instance is gathered with the function call in line five), a data adapter is fed. In the extension table layout, each extension is represented by one table. That means each property set (which contains the metadata regarding one extension) corresponds to one table in the persistent store (Jacobs and Aulbach, 2008). Therefore, no extensive mapping from an existent set of properties to a database table has to be done. The operations update, read, and delete work in an equivalent manner.

```
1   # if the instance is not persistent yet, "save" calls this method
2   # this method persists "self" (including all its extensions)
3   def create
4     values = self.instance_variable_get(:@attributes)
5     property_sets = @properties.deep_clone
6     adapter = MapperMania.adapter(MapperMania.default_adapter)
7
8     # store one property_set in order to retrieve a primary key
9     property_set = property_sets.delete_at(0)
10    new_id = adapter.create(values, property_set)
11
12    # set the gathered id in values to tie
13    # the extensions to this record
14    values[:@primary_key] = new_id
15
16    # store the rest the instance
17    property_sets.each do | ps |
18      adapter.create(values, ps)
19    end
20
21    # set the new id in the context of self
22    # this makes it accessible
23    @attributes[:@primary_key] = new_id
24  end
```

Listing 4. Implementation of create Method in a Domain Model

In order to guarantee the uniqueness of keys and a reliable mapping between data of a class and their extensions, the class is first persisted to the database in order to retrieve a unique key. This key is used later to bind the data of all extensions to the corresponding entry of the base class.

If the mapping schema is changed, only the operations create, read, update, and delete have to be reimplemented. These operations reside in the Model and Resource mixin and are therefore separated from the adapters.

This encapsulation also enables the usage of potentially multiple adapters and therefore data sources to compose a unified domain object. In Listing 4 the default adapter is used to store objects. Given more knowledge on the composition of business objects and an additional adapter, it is possible to query several data sources. This integration of other data storages happens completely transparent to the developer of domain logic, since the domain model either gets additional properties or does not change at all. It has to be augmented that such heterogeneity of several data storages with different access techniques and technology is strongly discouraged in a SaaS scenario. Since the software vendors have the infrastructure under their control, they can and should avoid the appearance of heterogeneity of any kind.

Data Storage Adapter

The data storage adapters encapsulate special characteristics of the corresponding data storages and provide a unified interface to the data mapper.

Listing 5 shows how the implementation of the create method of the SqLite3 (SqLite3, 2009) adapter looks like. This method receives the metadata of the properties and the corresponding values, the adapter is supposed to write to the data store in one request. An arbitrary list of properties can be passed to this method. The composition of a request works only on basis of the passed metadata and not on basis of the passed values. That way, the mapping rule processor can force the adapter to apply different mapping schemas. The metadata and values are then merged into an SQL (Date, 1986) query and sent to the database server. In order to retrieve the assigned key, SqLite3 requires sending an additional query. In this manner arbitrary data stores can be integrated into MapperMania. The operations read, update, and delete work in a similar way.

```
1   def create values, property_set
2     new_id = -1
3
4     property_set.properties.each_pair do | key, property |
5       key = '@' + key.to_s
6       property.value = values[key.to_sym]
7     end
8
9     stmt, bind_values = create_statement(property_set.table_name,property_set.properties)
10    execute(stmt, *bind_values)
11
12    with_connection do | connection |
13      stmt = 'SELECT last_insert_rowid() AS id;'
14      command = connection.create_command(stmt)
15
16      begin
17        reader = command.execute_reader(*bind_values)
18
19        while(reader.next!)
20          new_id = reader.values[0]
21        end
22      ensure
23        reader.close if reader
24      end
25    end
26  end
```

Listing 5. Implementation of create Method in an Adapter

Leveraging this encapsulation, different data access technologies can be integrated in MapperMania. Moreover, these adapters are set up during the start of the application and have to be passed to MapperMania in order to be used. MapperMania provides an adapter repository which stores them as key-value pairs and make them accessible in the whole framework.

CONCLUSION

Offering an ERP platform for SMEs poses new challenges in which multi-tenancy plays an important role. Mastering multi-tenancy turned out to be the key to provide a first-class SaaS application.

In order to leverage the positive effects of multi-tenancy, the sharing among tenants has to be improved. This paper proposes a mechanism which enables custom code modifications without changing a shared code base.

Furthermore, the area of tension between multi-tenancy and customization was described. To overcome the obvious contradiction between these two requirements, concepts were given and proven with an implementation. The implementation of the framework MapperMania gives developers the opportunity to completely abstract from multi-tenancy and therefore eases development of verticalizations. Thus, this framework supports the establishment of an ecosystem of partners which is absolutely essential to the success in the SME market segment.

Outlook

Considering the problems with implementing the adapters and with traditional object-oriented features such as inheritance, a data mapping framework might not provide the best results. In the long term the next logical step could be the implementation of a multi-tenant database.

Making business objects multi-tenant is the first step towards a customizable SaaS application for the SME market. Traditional concepts of on-premises software need to be reconsidered. Underneath these concepts is the static service layer. Further research is conducted on how changes in the business objects can be reflected in the service layer of the application.

REFERENCES

1. Müller, J., Krüger, J., Zeier, A. (2008) Improving Global Business of Medium-Sized Enterprises by Integrated Processes, *Proceedings of the 2008 IEEE Symposium on Advanced Management of Information for Globalized Enterprises (AMIGE'08)*, September 28-29 2008, 1-5.
2. Chong, F., and Carraro, G. (2006) Architecture Strategies for Catching the Long Tail, MSDN.
3. Fink, L., and Markovich, S. (2008) Generic verticalization strategies in enterprise system markets: An exploratory framework, *Journal of Information Technology*, 23(4):281-296.
4. Müller, J., Krüger, J., Enderlein, S., Helmich, M., Zeier, A. (2009) Customizing Enterprise Software as a Service Applications: Backend Extension in a Multi-tenancy Environment, *Proceedings of the 11th International Conference on Enterprise Information Systems (ICEIS 2009) (to appear)*, May 6 – 10, Milan, Italy, 2009.
5. Aulbach, S., Grust, T., Jacobs, D., Kemper, A., and Rittinger, J. (2008) Multi-tenant databases for software as a service: schema-mapping techniques, *In SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1195-1206, New York, NY, USA, 2008, ACM.
6. Zhang, L., Lee, M. K. O., Zhang, Z., and Banerjee, P. (2003) Critical success factors of enterprise resource planning systems implementation success in china, *In HICSS '03: Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS'03)* - Track 8, page 236, Washington, DC, USA, 2003, IEEE Computer Society.
7. Vilpola, I., Kouri, I., and Vaananen-Vainio-Mattila, K. (2007) Rescuing small and mediumsized enterprises from ineffcient information systems - a multi-disciplinary method for erp system requirements engineering. *System Sciences, 2007.,HICSS 2007, 40th Annual Hawaii International Conference on*, pages 242b-242b, Jan. 2007.
8. Somers, T. and Nelson, K. (2001) The impact of critical success factors across the stages of enterprise resource planning implementations. *Hawaii International Conference on System Sciences*, 8:8016, 2001.
9. Motwani, J., Subramanian, R., and Gopalakrishna, P. (2005) Critical factors for successful erp implementation: exploratory findings from four case studies. Comput. Ind., 56(6):529-544, 2005.
10. Hamilton, J., On designing and deploying internet-scale services, Technical report, Windows Live Services Platform, Microsoft, 2007.
11. Coffee, P., Busting myths of ondemand: Why multi-tenancy matters, http://wiki.apexdevnet.com/images/0/04/ Mythbust-MultiT.PDF, 2007.
12. Salesforce, salesforce.com, http://www.salesforce.com/, 2008.
13. Guo, C. J., Sun, W., Huang, Y., Wang, Z. H., and Gao, B. (2007) A framework for native multi-tenancy application development and management, *E-Commerce Technology and the 4th IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services*, 2007, CEC/EEE 2007, The 9th IEEE International Conference on, pages 551-558, July 2007.
14. Jacobs, D. and Aulbach, S. (2007) Ruminations on multi-tenant databases, In Alfons Kemper, Harald Schoning, Thomas Rose, Matthias Jarke, Thomas Seidl, Christoph Quix, and Christoph Brochhaus, editors, BTW, *volume 103 of LNI*, pages 514-521, GI, 2007.
15. Boyen, N., Lucas, C., and Steyaert, P. (1994) Generalised mixin-based inheritance to support multiple inheritance, *Technical Report 12, Vrije Universiteit Brussel*, 1994, vub-prog-tr-94-12.
16. Bracha, G. and Cook, W. (1990) Mixin-based inheritance, *In OOPSLA/ECOOP '90: Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, pages 303-311, New York, NY, USA, 1990, ACM.
17. Schaerli, N., Ducasse, S., Nierstrasz O., and Black, A. (2003) Traits: Composable units of behaviour, pages 327-339, 2003.
18. Muller, P.A., Fleurey, F., and Jezequel, J. M. (2005) Weaving Executability into Object-Oriented Meta-languages, *8th Int. Conf. on Model Driven Engineering Lanuages and Systems*, 2005.
19. Fowler, M. (2005) Language workbenches: The killer-app for domain specific languages?, 2005.
20. SqLite3, SqLite3, http://www.sqlite.org/, 2009.
21. Date, C. J. (1986) A guide to the SQL standard, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.