# Object-Relational Mapping with SqueakSave

Thomas Kowark Robert Hirschfeld Michael Haupt

Hasso-Plattner-Institut für Softwaresystemtechnik, Universität Potsdam

{firstname.lastname}@hpi.uni-potsdam.de

## Abstract

Object persistence is an important aspect of application architectures and development processes. Different solutions in this field evolved over the last decades and new approaches are still subject to research. While object-oriented databases become increasingly popular, the usage of relational databases through an object-relational mapping layer is still one of the most widely adopted techniques. However, most object-relational frameworks require a considerable amount of mapping descriptions between object models and relational database schemas. This additional layer has to be maintained by developers along with the object model itself.

In this paper, we present an approach to object-relational mapping that utilizes the introspection and intercession features of Smalltalk to free developers from manually creating those mapping descriptions. The presented framework analyzes the existing models and automatically deduces suitable database schemas. Thus, it aids development processes by neglecting the need for a separate mapping layer.

A detailed introduction of the programming interface is followed by a description of the framework's internal implementation details. Additionally, the performance of the framework is evaluated through a comparison against a comparable system for the same programming environment.

***Categories and Subject Descriptors*** D.2 [*Software Engineering*]: D.2.2 Design Tools and Techniques Object-Oriented Design methods D.2.11 Software Architectures Data abstraction

***General Terms*** Design, Experimentation

***Keywords*** Object-relational mapping, impedance mismatch, automatic schema creation

## 1. Introduction

Maintaining application data in persistent storage spaces is an inherent requirement of most applications. Especially the web applications that have evolved over the past few years need to handle steadily growing and evolving data schemes. While this requirement obviously has an impact on the complexity and execution speed of applications, it also influences their development processes.

One of the main criteria for the choice of a suitable persistence strategy is project scope. Enterprise applications rely on robustness, execution speed and scalability [3], whereas smaller projects additionally focus on the flexibility to quickly adapt to changes in the object model [2]. Thus, development teams need a persistence solution that does not impede their development process, but allows them to implement new features in a simple and straightforward manner.

In addition to project scope, decisions regarding the development environment and language also influence the choice between available persistence strategies. Especially dynamically-typed languages like Smalltalk vastly reduce turn-around and implementation times by offering a programming paradigm that embraces change of existing implementations [29] and strong meta-programming and reflective features. The latter, however, impose non-trivial challenges for the implementation of persistence management systems.

Today many persistence strategies are available [5, 11, 18, 24, 28]. Their underlying data storage technologies cover a wide spectrum, ranging from purely relational databases over relational databases enriched with object-oriented techniques, to completely object-oriented implementations. The ease-of-integration of those solutions into dynamic object-oriented applications differs strongly [15] as the mismatch between the paradigms founding the application development and the persistence framework varies in its extent [2].

A widely adopted solution within this field is the usage of relational databases along with an object-relational mapping (O/R mapping) layer that bridges the gap between an application's object model and the relational schema of the underlying database [1]. Generic O/R mapping frameworks cover a variety of aspects reaching from basic CRUD[1] functionality to more elaborate features like transaction processing.

[1] Create, Read, Update, Delete

However, most available systems require extensive meta-description of the object model in order to be able to perform the aforementioned tasks.

Such descriptions impose a considerable burden on application development. Each change of the object model implies an alteration of the description layer [21, 22]. Seamless integration of O/R mapping frameworks into applications is moreover influenced by the degree of intrusiveness into the existing object and programming model. A high degree of transparency of the underlying database structures and systems is desirable [20]. Still, existing implementations vastly differ in the extent of implementation detail exposure to the user. This includes query APIs that are not integrated into the chosen programming language as well as the need to alter inheritance hierarchies or even object layouts in order to store objects in relational databases.

Within this paper, we present a framework that uses the strong introspection and intercession capabilities of Smalltalk in order to free developers from the creation of extensive object model meta-description. Based on the objects created during application runtime the framework automatically deduces suitable database schemas that are also adopted whenever developers change their object models. The entire framework thereby remains non-intrusive in a sense that no changes to existing object models have to be performed and queries on the data space can be carried out by using the well know semantics of the Smalltalk collection protocol. By that, the system combines the technique of object-relational mapping with Smalltalk development paradigms and provides an object-oriented database like behavior within a relational-database access layer.

Squeak[2], an open-source Smalltalk dialect, has been chosen as the development environment for the implementation of the framework due to its focus on educational purposes and the availability on a variety of platforms. The proposed framework is named SqueakSave[3].

The first part of the paper presents the workflow of the integration of SqueakSave into an application. Following those usage descriptions, the architecture of the framework is discussed with a focus on implementation details of the main workflows. After the performance of the system is compared against a popular O/R mapping solution available for Squeak, the paper concludes with remarks about related work within the field of O/R mapping in dynamically-typed object oriented environments and an outlook about future extensions that could further improve the usability and performance of the framework.

## 2. SqueakSave

In the following, an introduction to the basic usage patterns of the SqueakSave O/R mapper is provided. A simple weblog example application accompanies the description in order to ease the understanding of basic features as well as more elaborated techniques, such as transactions or custom mapping descriptions.

[2] `http://www.squeak.org`

[3] `http://www.hpi-web.de/swa/squeaksource/SqueakSave.html`

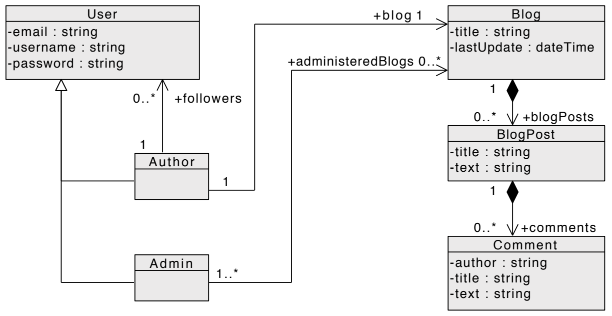

**Figure 1.** Class Structure of the Example Application.

```
SqsConfig subclass: #BlogExampleSqsConfig
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'BlogExample'

BlogExampleSqsConfig class>>#connectionSpecification
    ↑ SqsMySQLConnectionSpecification
        user: 'admin'
        password: 'password'
        database: 'blog_example_db'
```

**Listing 1.** Configuration Set-Up.

The class structure of the sample application is depicted in the UML class diagram [23] in Figure 1. It exhibits the most common structural challenges that O/R mappers have to handle within applications [13]: inheritance relationships and to-one or to-many associations. While the current section presents the integration of SqueakSave into the weblog application, the mapping of those structural details is the topic of Section 3.

### 2.1 Basic Persistence Mechanisms

A main requirement for SqueakSave is to provide straightforward persistence mechanisms in a very simple manner. Below, we present the steps that are required in order to set-up and use the framework for most basic purposes. This includes means to store objects within the chosen RDBMS and query for objects based on certain attribute values.

***Initial Setup and Configuration.*** For each class of objects that need to be persisted, developers have to set-up an instance of `SqsConfiguration`. Configuration objects include numerous properties that determine the behavior of the framework for the classes they apply to.

In order to register a configuration for the application classes, it is necessary to create a subclass of `SqsConfig`. The name of this subclass has to follow specific conventions to be recognized by the framework as being valid for a certain set of classes. To create a configuration for the entire application, the first part of the class category, which is normally subdivided by '-' characters [4], has to be the first part of the class name followed by the suffix *SqsConfig*.

In the simple use case of the blog example, only the class-side method `connectionSpecification` has to be implemented to return valid server access credentials. It determines which RDBMS is used as target storage for the respective objects. For each supported system, the framework provides a specialized *SqsConnectionSpecification* subclass. It provides standard values for port and hostname of common RDMBS server implementations such as MySQL or PostgreSQL. The only mandatory data are username, password, and the name of the target database. It is important that the user account provided for accessing the database has the privileges to create, alter, and drop tables, since SqueakSave constantly reorganizes the table structure according to changes within the application classes. The complete configuration class for the example configuration is depicted in Listing 1.

Following the aforementioned naming conventions, it is also possible to create different configurations for subcategories of the application by extending the category specific part of the class name prefix.

If the configuration itself has to be altered, it is possible to re-implement the `configuration` method on the class side of the configuration class. Additionally, the `configuration` method can be implemented on the class side of each application class, thereby providing the most fine-grained way of setting up configurations.

While it would be more compliant with object-oriented, and especially Smalltalk, principles to directly connect the class category with its configuration [17], this is not possible within Squeak, since the category is only identified as a string and not accessible as a first class object.

***Persisting Objects.*** Convention-based setup of configuration classes is essential to enable simple storing of objects. By extending the *Object* class, methods have been introduced that implement the data-modifying CRUD operations: creating, updating, and deleting objects. As a consequence of this 'monkey-patching'[4] any object, whose class is a subclass of *Object*, within the application can be stored and updated by sending it the `save` message. Since no database session or connection specification is passed as a parameter, this method relies on the previously set-up configuration objects and will trigger an exception if no configuration is available for the corresponding class.

Listing 2 presents the creation of an author object along with the associated blog. The `save` method will store the author object itself and the blog within the database and also create the one-to-one relationship between them.

Removing objects from persistent storage is possible by using the `destroy` method. It will remove the database rows

[4] `http://en.wikipedia.org/wiki/Monkey_patch`

```
author := Author new
   password: 'password';
   username: 'testuser';
   email: 'user@example.org'.

author blog:
   (Blog new title: 'My Blog').

author save.
```

**Listing 2.** Basic Object Storage.

```
(SqsSearch for: User) detect: [:aUser |
   aUser username = 'testuser']

(SqsSearch for: Author) select: [:anAuthor |
   anAuthor blog blogPosts size > 10 ]

(SqsSearch for: Blog) anySatisfy: [:aBlog |
   aBlog blogPosts noneSatisfy: [:aBlogPost |
      aBlogPost comments isEmpty ] ]
```

**Listing 3.** Query Examples - Emulated Collection Protocol.

corresponding to an object, and all references from other database tables to that object. Accordingly, destroying a user object within the sample application will also lead to a removal of the user from each `followers` collection it has been part of. While the database entries will be removed by the framework, the object itself remains unchanged.

***Object Query Interface.*** In addition to the modifying CRUD operations, a persistence framework has to offer means to perform queries on the persistent space. Since SqueakSave is built upon a relational database foundation, those queries have to be carried out as SQL statements. Integrating queries in such a way that standard language constructs can be used is an important feature with regards to the usability of an O/R mapper [9]. SqueakSave provides a query interface that does not rely on string-based query encoding, but instead emulates the Smalltalk collection protocol [8].

Object queries are usually sent to instances of *SqsSearch*. These objects must be initialized with a class; instances of this class and its subclasses will be returned by the query. Queries can be performed on each class residing within an image; however, a valid configuration for this class must be available.

Within the sample application, this behavior can be utilized to distinguish between authors and administrators. If searches are performed on the *User* class, they will return instances of *Admin* as well as *Author*. Performing searches on either of those classes individually, however, will only return their particular instances.

Listing 3 presents example queries that could be used within the blog example application. The first query performs a search for the user with the username *'testuser'*. According to the Smalltalk collection protocol, the `detect` method will only return the first user that is found within the database and trigger an exception if no such entry exists.

Query number two uses the aforementioned mechanism to narrow the set of possible search results down to special subclasses. The presented `select` method will find all authors that have a blog with more than ten blog posts.

The last query determines whether any object within a collection fulfills a given constraint. In this particular case the query will only return `true` if at least one blog exists where all blog posts have been commented at least once.

The messages sent to the query objects, such as `aBlog` or `aUser` are limited to accessor methods that are named exactly like the corresponding instance variables. Subsequent method invocations on the return values, such as collections, integers, or strings must be implemented within the respective classes of the SqueakSave framework (see Section 3.5).

In addition to the collection protocol emulation, SqueakSave offers convention-based dynamic query methods similar to those in other dynamic-language object-relational mappers such as GORM [28] for Grails[5] or ActiveRecord for Ruby on Rails [12].

```
(SqsSearch for: Blog) findByTitle: 'testblog'

(SqsSearch for: Comment)
    findByAuthor: 'author' andTitle: 'comment'.
```

**Listing 4.** Query Examples - Convention-Based Dynamic Finders.

The first query presented in Listing 4 depicts a simple use-case where instances of the *`Blog`* class have to be found by an exact match between the given argument and the current value of the `title` instance variable. The second search is an example for the concatenation of constraints. Concatenation keywords (i.e. 'and') adhere to SQL terminology. Thus, 'or' can be used as well within dynamic finders.

The aforementioned object-relational mappers allow for calling the dynamic finder methods directly on a class. In order to achieve the same behavior in Squeak, it would be necessary to either overwrite the `doesNotUnderstand` method within *`Class`*, or provide a means for application developers to integrate this implementation only within their model classes. This fine-grained integration could be achieved by providing an abstract base class that application classes have to inherit from. However, this kind of intrusion into the inheritance structure would not comply with the requirement to provide persistence as an aspect added to the application instead of being an integral part of it. A less intrusive technique is the usage of traits. They have been introduced in the Self programming language [30], and later been applied to Squeak [10] to provide a more fine-grained mechanism for reusing existing implementation details. By adding the *`TSqsSearch`* trait to any application object model class, queries can be performed as depicted in Listing 5.

[5] `http://www.grails.org/`

```
Blog findByTitle: 'testblog'

Comment findByAuthor: 'author' andTitle: 'comment'.
```

**Listing 5.** Query Examples - Convention-Based Dynamic Finders on Classes.

## 2.2 Customization

Utilization of the presented techniques to store and query for objects is sufficient to perform basic CRUD operations on application data. However, extensions are required for customizing the O/R mapping framework behavior, and for optimizing aspects of performance and robustness.

***Custom Configuration.*** The configuration object includes properties that define standard values for certain fields of the resulting database schema as well as architecture patterns that are used for the mapping of object-oriented structures to relational constructs.

Specialized configurations for subcategories and single classes are possible by implementing a `configuration` method in the respective configuration classes. The configuration object is available within those methods by calling `super configuration`. Attributes of objects referring to field names can be changed, e. g., to adhere to naming conventions of other O/R mappers, or to solve naming conflicts.

Altering the configuration can also be used to fine-tune framework behavior. It is possible to define whether instance variable accessor methods or object introspection mechanisms should be used to access instance variable values by setting `useInstVarAccessor` to either `true` or `false`.

While the framework by default alters table structures and association types only after developers confirmed those changes, the `warnOnAlteration` attribute can be set-up to disable the according warning dialogs.

When the object model is finalized and mapping update functionality is no longer required, the introspection behavior should be disabled in order to improve the overall performance of basic persistence operations. The `environment` attribute of the configuration can therefore be set to the value '#production' instead of its default value '#development'.

***Session Usage.*** While the implementation of SqueakSave frees users from the need to utilize an explicit session object to store, retrieve, and delete objects, some more advanced functionality is available only by using instances of *`SqsSession`*. Session objects can be retrieved from the singleton instance of the *`SqsConnectionManager`*. It caches the sessions on a per-thread basis. Thus, requesting a session for a certain configuration, class, or category will always return the same object within a single thread of control. The

```
sessionManager := SqsConnectionManager getInstance.
session := sessionManager
    sessionForClass: Blog.
session := sessionManager
    sessionForCategory: 'BlogExample'.
session := sessionManager
    sessionForConfiguration: aCustomConfiguration.
```

**Listing 6.** Possible Ways to Retrieve Session Objects.

```
transactionalBlock := [
    testuser email: 'newmail@example.org'.
    testuser save: session.
    testuser password: 'newPassword'.
    testuser save: session.
].

session
    inTransactionDo: transactionalBlock
    ifError: [ testuser rollback ].

"alternatively"
session startTransaction.
transactionalBlock value.
session commitTransactionIfError: [ testuser rollback ].
```

**Listing 7.** Transactions within Sessions.

different possibilities to get the current session for the sample application are depicted in Listing 6.

With the session object, it is possible to perform transactions and define the intended behavior upon transaction failures. If the SqueakSave session is, for example, stored within a Seaside[6] session object, and all data manipulation operations are performed by passing the session as an explicit parameter, transactions can even span the entire life cycle of web application usage by a single user. Transactions do not have to be performed by defining a block-closure for the transactional behavior and one for the rollback case, but it is possible to explicitly start and commit them via the respective methods of the session protocol.

Listing 7 depicts the two possibilities by using an explicit session object that has been retrieved like shown in Listing 6. The `rollback` method will set the instance variable of the user object back to the pre-transaction state.

***Performance Optimization.*** The database schemas created by SqueakSave follow the basic patterns described by Fowler et. al [13] - single, concrete, and class table inheritance. However, not all of those patterns may be suitable for each object model. Especially deep inheritance hierarchies can create performance problems, if they are mapped to a single table. Additionally, an abstract base class for all application classes should be ignored for persistence purposes, since each subclass instance has to be saved within the base class table, as well (class table inheritance), or all application objects will reside within the same table (single table inheritance).

[6] `http://www.seaside.st`

```
newBlog := Blog new;
    title: 'New Blog'.
newPost := BlogPost new;
    title: 'New BlogPost'.
newComment := Comment new
    title: 'New Comment'.

newPost comments add: newComment.
newBlog comments add: newPost.
testuser blog: newBlog.

testuser flatSave.
testuser save.
testuser saveToLevel: 2.
testuser deepSave.
```

**Listing 8.** Different Save Levels of SqueakSave.

SqueakSave also offers means to control the object graph traversal depth required to store or update objects. Within the example that is presented in Listing 8, the consecutive usage of the different methods that enable this behavior will gradually store more associated objects of the user object. While `flatSave` only stores direct attributes, `save` also includes the blog of the user into to storing process. With `saveToLevel:2` the blog post is considered, since two references have to be followed from the user to those objects. The final call of `deepSave` stores every object reachable from the user object and only stops upon cyclic dependencies or if no further references are detected.

***Custom O/R Mapping Descriptions.*** While SqueakSave mostly hides the creation and handling of O/R mapping descriptions, they are not only kept in memory during persistence operations but are also stored within the image for later usage. The format of this persistence is defined by the chosen description handler class. This can be altered within the configuration object itself. The standard description handlers utilize the internal format of the meta-descriptions and simply serialize the corresponding objects. However, custom mapping descriptions, such as pragmas or XML documents can be generated as well, if the corresponding description handler classes have been implemented. Due to this fact, the techniques to mark descriptions, or parts of it, as being manually maintained, differ between the description handler implementations.

Regarding the standard description handler, each description includes a `manuallyMaintained` flag that indicates whether it is maintained by users or not. If this flag is set, automatic updates will not alter the particular description. However, if the custom description requires changes to the database schema, they will be carried out by the framework.

A variety of options can be altered within the mapping description for particular instance variables. This includes trivial values, such as the column name or the SQL type of

```
AccountData class>>#sqsDescrUsername
    ↑ SqsColumn new
        manuallyMaintained: true;
        columnName: 'name';
        sqlType: #varchar:20;
        linkedAttribute: #username.
```

**Listing 9.** Custom Mapping Description.

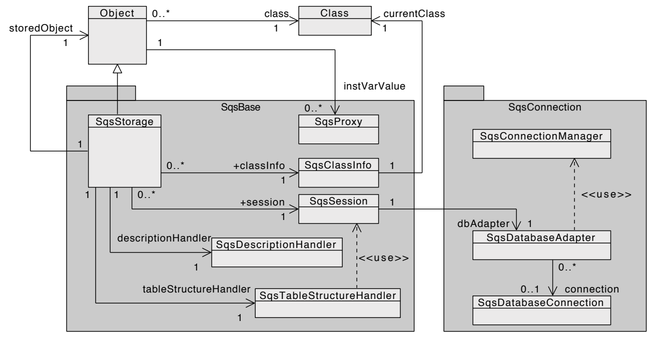

**Figure 2.** Overview of SqueakSave System Classes

the column, but also more advanced features like foreign-key constraints. Additionally, it is possible to alter the name of the table, that is created for each class. Listing 9 depicts a custom configuration for the *username* field of the account data.

### 2.3 Summary

The preceding presentation of the usage workflow of SqueakSave has demonstrated, that the requirements regarding simplicity of usage as well as customizability as a means to increase interoperability, have been fulfilled. It becomes apparent that only minimal configuration is necessary, in order to add persistence in a very transparent manner to an existing application. While the API of SqueakSave may not comply with every other available solution, and thus changes to the source code might have to be carried out, this does not necessarily decrease the ease-of-integration. It is generally advised to encapsulate database access functionality in a separate layer between the application and the persistence framework. Within this layer the presented CRUD-functionality can be implemented in a very intuitive manner.

## 3. Framework Architecture

The usage workflow described in the preceding chapter is realized by the core classes of the SqueakSave framework. They are depicted in a simplified manner in Figure 2, i.e., without the inclusion of concrete subclass implementations.

### 3.1 Storage Wrapper Class

Enriching objects with capabilities that have not been implemented within their respective class definitions can be realized by utilizing a number of standard patterns. As existing class definitions shall not be altered, the SqueakSave framework relies on the *SqsStorage* class as a decorator [14] that handles persistence-related operations such as storing, updating, or deleting objects.

Accordingly, calls of `save` or `destroy` will be internally delegated to an instance of *SqsStorage* instead of being handled completely by the target objects themselves. For each object that is present within the image, a unique *SqsStorage* instance is created on demand. Due to a caching mechanism that is utilizing weak references [16], the respective instances are only available as long as the base object is not subject to garbage collection.

In addition to the decorator, the framework will also assign a unique object id to each persisted object. Those unique identifiers, that are usually generated by the respective RDBMS, are required to couple an object to its database representation and, accordingly, enable references between objects on the database level [1]. The ids are stored as an instance variable of the decorators within the image and in a primary key column within the database.

The decorator is connected to the current database session and by that has access to the corresponding configuration for the decorated object. The configuration determines the classes of the `descriptionHandler` and `tableStructureHandler` instance variable variables.

### 3.2 O/R Mappings: Creation and Update

The description handler is responsible for creating mappings between objects and their database representations. It does not create the underlying database schema but analyzes the given objects using introspection and creates detailed descriptions for the current values of an object's attributes.

For most basic data types, such as strings or integers, the mapping to relational constructs is straightforward. The suggested column names are simply deduced from the instance variable name and the types are pre-defined within `sqsType` methods on the class side of the respective classes. This methods return a SqueakSave internal string representation of the according SQL type. For types with variable length the mappings are additionally enriched with the information about the current length of the respective object.

Information about complex attributes—objects that cannot be mapped to simple SQL types but require a separate table structure—is additionally tagged with the class of the respective object as well as a generic description of a foreign-key relation to the database table for that particular class. For attributes holding collections of objects, the type of the collection, the class of the collection index, and the class of the included elements have to be determined.

All this information is persisted in the format specified by the corresponding description handler. Upon every save of an object the description handler has to determine whether changes to the relational structure would be necessary by examining each instance variable for differences compared to the previous version of the description.

Alterations can become unavoidable in a variety of scenarios. Most obviously that is the case if the class of an assigned value has changed. However, not every object class change requires a database structure change. Certain types comply with each other with regards to their database representation. Within the example application, this behavior could be observed if an *Admin* object is the current value of an attribute that was previously pointing to general *User* objects. For collections, it is also necessary to determine whether the type of the collection itself has changed since indexable collections like an *OrderedCollection* or a *Dictionary* would require the storage of the index, whereas a *Set*, for example, would not require such a field.

Depending on the specified configuration, the framework issues a warning dialog before changing the descriptions. If developers decide to not allow the requested changes, the storing procedure is aborted.

## 3.3 Table Structure Adaption

After the mapping descriptions have been updated, the *SqsStorage* decorator passes control to the table structure handler. It translates the general attribute descriptions to representations of actual relational constructs (tables, columns, or constraints) and thus builds a generically traversable abstraction from the actual table structure. Each such table object can have a number of columns, foreign key constraints, and child tables. In addition, each child table also includes a reference to its parent table.

In a straightforward case, however, the structures created from the descriptions are rather simple. Depending on the inheritance mode specified within the configuration, all attributes reside within the same table (single table inheritance), or a separate child table is created for each subclass (class table inheritance). Within those tables, a column with the previously determined SQL type is created for each simple attribute. For complex attributes, the handler will also create a foreign-key constraint that guarantees the referential integrity of the reference to the table of associated objects.

***Collection Mapping.*** Collections of objects are always created as join tables, and not like in other O/R mappers in case of one-to-many relations as foreign keys within the table of the referenced objects. This is a direct consequence of two problems. The first one is the distinction between one-to-many and many-to-many relations through reflection. While it would be possible to detect those relations, implementing this feature has proven itself to be too time consuming during program execution. Not only would the framework be supposed to follow all references pointing to objects within a collection, until one is found that has more than one reference to it. But, additionally, database queries would be required to check if references exist that are not currently present within the application's object memory.

The second problem is the inversion of the logical association direction from the object model to the relational structure [21]. Instead of the collection owner pointing to the values of the collection, elements within that collection would reference their owner. This fact is also problematic regarding object usage within many collections in different classes or instance variables of the same class. It would be required to add a new table column for every reference to those objects.

The created join tables contain a field referencing the table entry of the collection owner and another column pointing to the respective object within the collection. Additionally, an order field is introduced if the application uses ordered collections. This field is created with the type of the index value of the collection. To map an *Array*, for example, the index field would be of type INTEGER, while a string-indexed dictionary would require a VARCHAR type. If the collection only includes simple values, the reference field to collection elements will be replaced with a field of the respective type that directly stores them within the join table.

***Structure Updating.*** If the table structure already has been created, the table structure handler compares a cached version of the class table with the one created from current descriptions. The *SqsTableChanges* class is capable of comparing two tables and extract all columns, whose names or types have been altered. Additionally, it detects added and removed columns and foreign key constraints. All required changes are subsequently carried out on the database.

Since this process is highly sensitive to interference with similar operations carried out by other processes, a semaphore guards the entire structure update and creation workflow. While this might diminish the overall system performance, it is necessary to keep the cached table structures and, accordingly, the database schema in a consistent state.

Finally, after the table structure has been altered to the required schema, the description handler inserts the values into the corresponding tables.

## 3.4 Supporting Workflows

The previously described procedures are sufficient for the basic implementation of O/R mapping and table structure creation and updates as well as insertion of the actual values into the database. However, more elaborated workflows are required to improve the mapper's performance or handle special circumstances, such as cyclic dependencies.

By tightly coupling decorator instances to decorated objects, it is possible to cope with recursive calls of the `save` method. Decorator instances will only try to store associated objects if the current object has not already been processed in the present operation. A flag is set upon first traversal, and if cyclic references lead to an object again, only changes to instance variables and owned collections will be examined.

Decorators also create a simple representation of the state of the decorated object upon each `save` call. This so-called instance variable value map enables the framework to quickly determine whether an object has changed at all and if so, which variables have changed. Unchanged variables will be ignored during mapping description updates and also not be part of the 'UPDATE' statement issued on the database.

***Database Connection Handling.*** Database adapters encapsulate SQL query generation according to the specifications of the respective RDBMS. To execute those queries, adapters rely on *`SqsDatabaseConnection`* instances. These conceal differences between the connection objects supplied by the different database access drivers.

The physical database connection is obtained by the database adapters only when required, and dropped whenever queries have been executed successfully. While connecting and disconnecting to the server upon each request would have simplified the implementation, it is not a viable approach with regards to performance. Login procedures on database servers are rather costly in comparison to execution times of smaller queries. Therefore, SqueakSave implements a centralized connection pool. This pool is maintained by the singleton *`SqsConnectionManager`*, and due to a *`SharedQueue`* implementation also thread safe. Each adapter that requires a database connection has to utilize the connection manager and either get it instantly, or whenever a connection is returned to the queue by another adapter. The shared queue guards the insertion and retrieval processes. Hence, it is guaranteed that each connection is only assigned to one adapter at a time. All adapters that have to wait for a connection are also waiting for the semaphore to become available and, accordingly, race conditions are prevented in this scenario, too.

While this standard behavior is suitable for most basic operations, it obviously cannot be used during transactions. Therefore, each database adapter is aware of its current transaction state and does not return connections to the queue while a transaction is in progress.

## 3.5 Query Generation

The following section provides a detailed explanation of the SQL query generation from method invocations on the language-native query API.

***Collection Protocol Emulation.*** The implementation of the collection protocol emulation for object queries is based on the work of W. Harford and E. Hochmeister, who have implemented a quite similar system for the ReServe project[7]. While the basic implementation allowed for simple queries on directly associated attributes of objects, it has been enriched with the capabilities to define query conditions on associated collections and directly associated objects to a much deeper level within the object graph structure.

In order to analyze the block-closures that are passed as arguments to the respective collection methods, SqueakSave utilizes the *`SqsQueryValue`* classes depicted in Figure 3. Each of those classes imitates the protocol of basic system

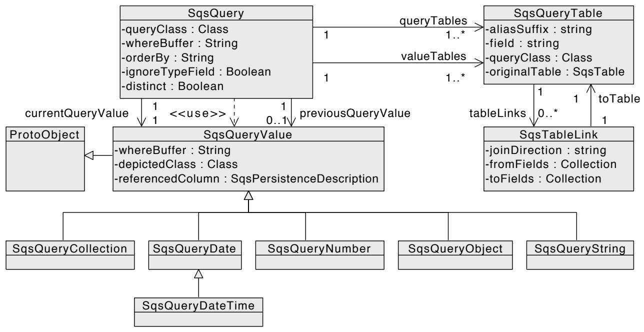

**Figure 3.** Collection Protocol Emulation Classes

classes such as *`Integer`* or *`String`*. But instead of delivering the result for each operation, the methods gradually fill the `whereBuffer` attribute with the SQL equivalents of the respective operations. Listing 11 presents the SQL WHERE statement that is generated for a sample query (Listing 10).

```
query := (SqsQuery on: BlogPost) analyze: [:aBlogPost |
    aBlogPost text size > 100].
```

**Listing 10.** Language-Native Query Before Translation

```
'WHERE CHAR_LENGTH(blog_posts.text) > 100'
```

**Listing 11.** Generated SQL WHERE Statement

Complex objects, that cannot be directly mapped to an SQL type are depicted by instances of *`SqsQueryObject`*. Each method sent to those objects is analyzed with regards to the database columns representing the corresponding attribute. If such a column exists, the where buffer is enriched with a unique identifier consisting of the according table and column name. If columns refer to rows in different tables (i.e., foreign key relations), this scoping is performed by *`SqsQueryObject`*s, too. Upon each scoping to another table, the table names are being aliased with a unique suffix, that allows for self-referencing foreign key handling.

In addition to the WHERE statement creation, the system also conglomerates the tables that are important to the query within *`SqsQueryTable`* objects. They include a unique suffix and a reference to the *`SqsTable`* object, that serves as a meta-description of the database table structure. Additionally, a number of links to other tables can be added to a query table, in order to represent joins that have to be performed for queries. During the final steps of query generation, those query tables are connected to form the FROM part of the SQL query. Tables, whose values have to be returned from a query, are stored in the `valueTables` collection of an *`SqsQueryObject`*.

This generic analysis of block-closures allows the framework to handle table structures for class and single table inheritance and the nesting of constraints, e.g., for sub queries on collections that are owned by query objects, without any explicit distinctions between the different table models.

[7] `http://www.squeaksource.com/REServe.html`

```
[:aComment | (aComment author = 'author') &
    (aComment title = 'comment')].
```

**Listing 12.** Block-Closure Generated from Dynamic-Finder Method.

***Convention-Based Query Methods.*** The implementation of the convention-based dynamic query methods is also based on the collection protocol emulation. Therefore, the finder methods are analyzed for the occurrence of attribute names and the respective values. This is performed within a re-implementation of the `doesNotUnderstand` method that handles calls of undefined methods on objects. The method checks whether the first part of the selector either matches *find* or *findAll*. If either of those strings matches the beginning of the given method selector, the remaining parts are scrutinized for their compliance with instance variable names of the respective search class. Finally, the algorithm determines the logical operators that are implied by the method name.

Afterwards the framework creates block-closures depicting those constraints and concatenates them with the chosen logical operators. The block-closures are generated by utilizing the previously extracted strings from the method selector name and the arguments passed to the dynamic finder method. The values are especially important in this case, since they have to be translated into a string. Complex objects, for example, require the inclusion of their object id into the query string, while simple types such as dates or strings need to be escaped to be properly parsed by the Squeak compiler. Therefore, the *`SqsSearch`* class maintains a dictionary with the respective methods, it has to call for certain types of objects. If the string representation has been successfully generated, it is passed to the *`Compiler`* that generates executable bytecode for the required block-closure.

This block-closures will be then forwarded to an instance of the *`SqsQuery`* class, that analyzes them as described previously. Listing 12 depicts the block-closure created from the second dynamic finder method presented in Listing 5.

***Object Proxies.*** For performance and framework internal reasons, instances of *`SqsProxy`* are inserted into query results instead of directly associated complex objects or collections. There are dedicated proxies for directly associated objects and those representing collections.

Proxies for directly associated objects like a user's blog in the sample application are necessary to avoid an eager loading of the entire object graph upon the creation of query results. The proxies are initialized with all information required to trigger loading of the depicted object if the application accesses them. All calls to proxy objects, except for those defined on *`ProtoObject`*, are delegated to the loaded instances. Thereby, proxy insertion remains transparent to framework users and the proxies could also be removed once the depicted object is present within the image.

Collection handling requires a different approach to proxy insertion. While the aforementioned objects only serve as placeholders, collection proxies are essential to detect changes in collections. Therefore, before each `save` call and after loading an object as the result of the search query, an instance of *`SqsCollectionProxy`* is inserted instead of the original collection. In addition to loading all objects that are part of the original collection, those proxies also create and maintain an internal map of the collection objects. This allows the framework to detect added, displaced, and removed objects in a collection. Hence, after each successful `save` call, the collection map will be updated, and if the object referencing the collection is saved again, all changes that happened up to this point will also be reflected within the database.

***Object Caches.*** In addition to using caches for object id storage without object model or inheritance structure alteration, query performance optimization also requires this feature. To avoid rebuilding objects that already are query results, or have been instantiated just recently, it is necessary to maintain an additional cache. It has to return pre-built instances identified by their class name and object id.

While caching all available objects could improve the performance of query result creation, a trade-off between the memory footprint of the framework and the performance gain induced by result caching has to be made. Therefore, the cache size is limited on a per class basis to a configurable number of entries and makes it possible to implement different cache sizes for each application.

## 3.6 Framework Extension

A central requirement for the development has been the extensibility of the framework with regards to the adoption of newly available database management systems and the implementation of custom O/R mapping flavors. Therefore, the classes responsible for realizing the corresponding behavior have been implemented in ways that ought to simplify the development of custom framework extensions.

***Custom Object-Relational Mapping Descriptions.*** The *`SqsDescriptionHandler`* serves as an abstract base-class, that defines the methods, which are crucial to the implementation of custom description handlers.

Only two methods have to be implemented in order to create new mapping description handlers. `sqsDescriptionFor:` returns the meta-description of the O/R mapping for an instance variable of the object that is subject of currently performed persistence operations. While this description can be stored in arbitrary formats, the method always has to deliver instances of *`SqsPersistenceDescriptor`*. This translation might be costly with regards to time consumption, but developers could avoid performance problems by caching the SqueakSave-internal format or persisting it by utilizing the standard description handlers.

The second method that needs to be implemented is `createDescriptions`. It is called during the storing process and, since the description handlers have full access to the decorator of the persisted object, requires no additional parameter. While it would compromise the self-configuring nature of SqueakSave, to not create or update mapping descriptions, custom description handlers that should only supply reading abilities can waive this implementation.

***Database Adapters.*** An obvious extension point for an O/R mapper are adapters for different RDBMS. They implement the generation of the SQL queries depicting certain database operations. In order to provide a custom adapter, two steps are mandatory for alleged extension developers.

The first one is to create a subclass of *`SqsConnection`* that implements some basic operations to control the state of the actual database connection and execute queries on them. The connection control methods are required in order to automatically create new connections within the connection-pool. Therefore the `init`, `close`, and `isAlive` operations have to be implemented. In addition to the query execution, the framework also requires means to convert the query results from the client-internal format into a general one, that can be handled by SqueakSave adapters.

While it is necessary to re-implement those methods for each adapter facilitating a native client implementation, it would be possible to utilize an open standard interface that provides the same access methods, regardless of the underlying database. This includes connectors like ODBC[8] or OpenDBX[9]. However, the setup of those two solutions requires not only the installation of respective clients for Squeak, but additionally the installation or even compilation of platform-dependent libraries within the operating system.

The methods within the protocol of *`SqsDatabaseAdapter`* that have to be overridden in order to provide a working adapter implementation for a certain RDBMS are rather difficult to be determined. This is mainly a consequence of the custom extensions to the SQL-standard implemented by different RDBMS vendors. The basic implementation within SqueakSave, however, strives to implement almost all operations according to the SQL standard. This should minimize the number of methods that have to be overwritten.

### 3.7 Summary

Main requirements for the implementation were the realization of automatic updates, language-native queries, and extensibility of the framework. Above, necessary design decisions for the implementation of this behavior have been presented. Automatic updates are implemented by a copious algorithm that covers almost all possible changes to object models and therefore dependably and only updates existing mapping descriptions if necessary.

Language-native queries have been implemented by a block-closure analysis system that can handle deep object graph structures and standard operations on simple data types as well as accessor methods on complex objects.

Extension points are also available for all designated components of the framework and provide meaningful presets for the implementation of custom description and table structure handlers, as well as database adapters

## 4. Evaluation

The main focus of the implementation of SqueakSave is the support of fast-evolving object models and the development of a generic architecture that allows for extension of the available description systems, table structure handlers and database adapters. However, performance is an important aspect of each persistence management system [2]. Accordingly, the implemented framework has to be evaluated with regards to both aspects. The following section provides benchmark results for SqueakSave in comparison to another O/R mapping framework for the same development environment. Additionally, the production and development modes are compared and conclusions are drawn regarding performance bottlenecks and possible optimizations.

### 4.1 Performance

Numerous benchmarks exist to measure the performance of object persistence technologies. The BUCKY [7] or the BORD benchmark [19], for example, are especially designed to analyze the performance of object-relational systems. Other approaches, like the OO7 Benchmark [6], have been developed to provide objective measurements for any kind of object persistence, without any special focus.

One of the requirements for the implementation of SqueakSave is to provide persistence in a transparent manner. Therefore, the OO7 Benchmark is utilized for performance measurements. The implementation used for this comparison is based on the Java version[10] of the original benchmark, which was written in C. It was ported to Java to compare the performance of object-relational mappers and object-oriented databases [31].

Measurements have been carried out on a 2.4 GHz Intel Core 2 Duo Macbook with 4GB RAM and Mac OS X 10.5.6. PostgreSql version 8.3 has been used as the underlying RDBMS. Each benchmark was run 100 times; measurement results represent the median of all retrieved timings.

### 4.2 Comparison with other Object-Relational Mappers

Since platform specific limitations and performance bottlenecks, such as overall inferior execution speed or subpar implementations of viable system classes, impede objective

[8] `http://support.microsoft.com/kb/110093`

[9] `http://www.linuxnetworks.de/opendbx`

[10] `http://sourceforge.net/projects/oo7/`

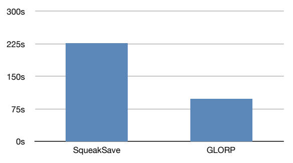

**Figure 4.** Benchmark Database Creation Times

measurements, a meaningful comparison can only be performed against a comparable system implemented within Squeak: The generic lightweight object-relational persistence framework (GLORP) [18].

In addition to pure performance comparisons of aspects like object creation after queries, it is also interesting to see how the different implementation paradigms of GLORP and SqueakSave compare to each other. SqueakSave requires explicit save operations to store or update objects, while GLORP is transaction based. Accordingly, the transaction based frameworks are able to accumulate all operations on the data and perform them, if possible, in bulk SQL statements. The benchmarks will identify scenarios where this behavior is beneficial with regards to performance.

The PostgreSql Client 1.0 was used in a Squeak 3.10 image running on the Squeak VM version 3.8.18. SqueakSave was used in revision 107, and GLORP in version 0.4.169. To further avoid influences on the measured timings, both systems were set-up to their respective production environment, i.e., SQL statement logging and other debugging features have been disabled.

The benchmark consists of two parts. The first one performs a number of plain search queries on the created object space and measures the timings for each of them. The second part traverses object hierarchies from distinctive starting points and performs some alterations of the respective objects. In addition to those standard parts, database creation times have been examined, as well. While the insertion of such an highly intertwined and large object graph might not reflect everyday usage patterns of object-relational mappers within applications, it is an indicator for alleged performance bottlenecks and optimization potentials.

The overall database size of the benchmark can be configured in four orders of magnitude. Each of them increases the amount of stored objects and connections between them. The third-largest version of the benchmark was used, since it reflects the intended application area for the SqueakSave framework in terms of database usage. It includes approximately 10.000 atomic parts with 30.000 connections and thus reflects the database payload of small to mid-sized applications.

Figure 4 presents the overall creation time for the database schema that is required to perform the OO7 Benchmark. It is evident that GLORP outperforms SqueakSave by far. This is mostly a consequence of the ability to delay the insertion of objects into the database and perform them at a later point in a bulk operation. Thereby, instead of numerous single

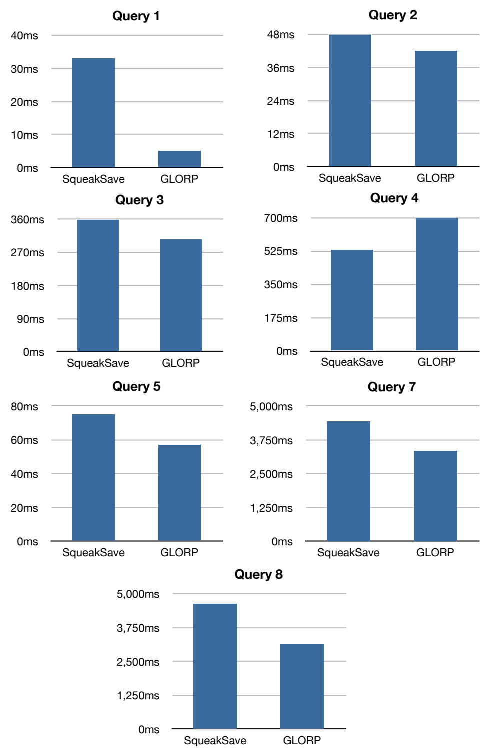

**Figure 5.** Benchmark Query Times

queries, only a few large ones are carried out and, accordingly, the overall execution time decreases. While this technique obviously could improve the performance of SqueakSave within such insertions, the decision to only provide direct save methods has been made with regards to API simplicity and not execution speed.

***Query Performance.*** The queries performed during the OO7 benchmark continuously increase in terms of complexity and result count. A description of the query contents is available in the paper that describes the original benchmark, as well as in the comparison carried out by Zyl et. al.

Query times presented in Figure 5 show that, regarding query performance, GLORP is generally faster than SqueakSave. The large difference in the first query, however, is not a result of superior query performance, but a consequence of optimistic caching. Instead of performing the query on the database, results are delivered directly from the cache. While this obviously increases query performance, it is also error-prone. Had the respective object been removed from the database in another session, the query would return an object that no longer exists in persisted space.

In all queries, except for the aforementioned one, differences between SqueakSave and GLORP are in a range of about 10–20%. The slight advantage in query four is a consequence of more efficient join table handling, since the generated SQL statements are almost equal, except for some minor differences in created table and column alias names.

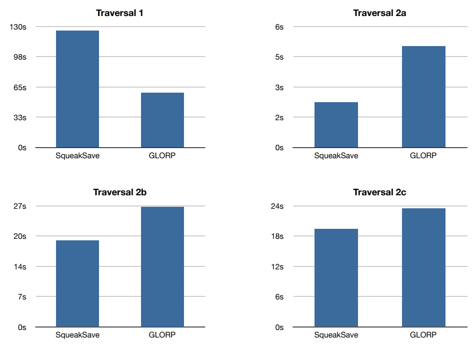

**Figure 6.** Benchmark Traversal Times

Unfortunately, the benchmarks reveal the tendency of an increasing distance between the two frameworks for expanding result sets. In queries seven and eight, the previous gap becomes vastly larger.

Concluding the query performance review, it can be stated that SqueakSave still has potential for optimization. While the difference for small result sets is minor and might be improved by smarter caching mechanisms, handling large result sets still remains an issue.

***Traversal Performance.*** The chosen traversal measurements of the OO7 benchmark all follow the same pattern. They start at the generated modules and navigate from the design root down to the atomic parts. With each traversal the depth of navigation through the object graph increases and, additionally, the last two also alter some data within the atomic parts. Traversal 2c not only changes those values once, but three times.

The other available traversals have been omitted, since they iterate through all characters of document texts and accordingly do not provide any insights into traversal speed, but only string operation performance.

Traversal benchmarks have been run independently from previous database creation and query tests. Those would have lead to extensive caching of the object graph and, therefore, could not reveal deficiencies within the loading of associated objects. For subsequent traversals, however, object caches have not been cleared in order to analyze the overall traversal performance and the caching of previously obtained results within one benchmark run.

The results depicted in Figure 6 unveil that only on first time object graph traversal, SqueakSave suffers from the currently missing support for eager loading of associations. Hence, the associated objects for each of the sub parts have to be obtained within multiple queries and can not be loaded in advance by a single one. The subsequent traversals, on the other hand, show that the huge disadvantage of SqueakSave turns around completely. This is a consequence of SqueakSave's caching mechanism, that gradually fills the central object cache during the first traversal. Hence, the entire object graph resides in memory for the second run. While the performance obviously improves because of that mechanism, the same coherence problem mentioned with regards to GLORP's first query result apply here.

The traversal times in the following tests obviously increase since the sub elements of the model are not only being traversed, but also updated. Therefore, it was expected that the advantage of SqueakSave slightly diminishes. However, the traversal times in those tests still show, that for the traversal of previously loaded object graphs SqueakSave seems to be a more efficient solution than GLORP.

The results have shown that SqueakSave, despite its automated mapping features can compete with existing O/R mapping solutions in terms of query and traversal performance. Especially, the caching mechanism makes SqueakSave a viable solution for sequential object graph traversals. The slow insertion times within large data-sets could be diminished by implementing a technique similar to the one introduced by GLORP. Special attention in future versions of the implementation has to be paid to the handling of large result sets, since they obviously impact the performance in a more than linear manner.

## 4.3 Development vs. Production Environment

The automatic creation of object-relational mapping descriptions is the main feature of SqueakSave. Due to the reflection mechanisms used to create this behavior, performance is obviously an issue that has to be examined closely. Therefore, the OO7 benchmark suite has been performed in development and production mode. The following results will reveal fields of usage where the automatic mapping behavior has a negative impact on the overall system performance, but also identify scenarios that are not affected by it. Additionally, insights into potential optimization points will be gained from those considerations.

Image 7 depicts the creation time for the small and tiny database layout. It can be clearly apprehended that the inspection of every object that has to be stored within the database slows down the overall performance. This is not a very surprising fact, since not only does the framework inspect each object, but also occasionally writes new descriptors to the image. Additionally, it has to check for and, if necessary, execute changes to the database schema. The performance degradation also seems to remain constant between the different benchmark scales, which implies that the table and description creation and updates have a much smaller impact on the performance, than the constant introspection measures. Obviously, after a very short period of time, no more alterations of the two models are necessary, and thus the difference between the two modes grows linearly.

While this slow-down might seem too high to be tolerated, developers should have to take into consideration that creating the scale 1 data model suffices to generate a valid database schema, that can be consecutively used to create the data-structures for the small or even bigger benchmarks. This, and the fact that the object-model can be developed in-

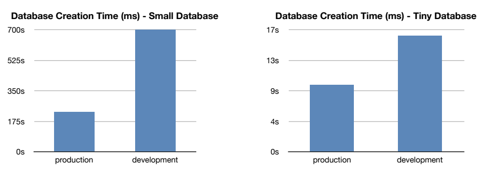

**Figure 7.** Benchmark Database Creation Times for SqueakSave Modes

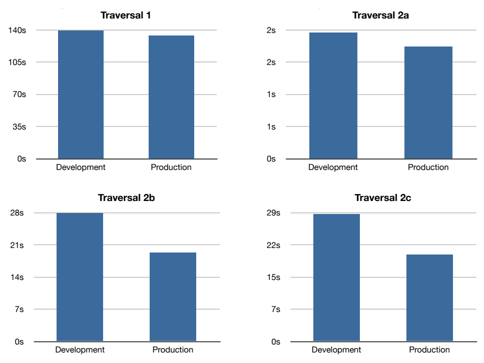

**Figure 8.** Benchmark Traversal Times for SqueakSave Modes

crementally without the necessity to alter database structures explicitly, relativizes the obvious performance impact.

Query performance does not differ between the two modes, since the synchronization between object model and database representation only takes place during object saving and, accordingly, does not affect search queries.

During traversal measurements, however, the previously observed differences still apply (see Figure 8). While the first traversal is barely affected by the current execution mode, changes to the object model (i.e., Traversals 2b+c) are performed much faster in production mode. It is therefore necessary for developers to thoughtfully utilize this feature if performance is important. Especially the role-based choice of the framework mode can provide a viable means for the balance between execution time and object model flexibility.

### 4.4 Framework Profiling

The benchmark implementation and execution provided a solid foundation for profiling the framework under a non-trivial workload. A couple of conclusions could be drawn, that can be incorporated into future framework upgrades.

- Much time of storing and query execution has been spent on automatic retrieval of configuration objects from the respective configuration classes. This is a direct consequence of Squeak's not incorporating categories as first class objects, and thus a time-consuming lookup for the respective classes has to be performed.
- Storing object ids in distinctive caches does not vastly affect execution speed. However, upon large scale operations, such as the creation of the benchmark database, the impact remains perceivable, since the according caches also grow with the number of in-memory objects.
- SqueakSave's current handling of large result sets suffers from the creation of ineffectively sized collections. While they provide a simple approach to the generation of objects from query results, their traversals are not optimized if the size exceeds certain values. Therefore, smarter algorithms have to be developed that utilize the Squeak-internal limits for efficient collection handling by splitting large result sets into smaller portions.
- Fine-grained `save` operations provide a viable means for controlling database insertions and updates. However, to accommodate larger object models or collections of objects that have to be inserted, they perform too many small queries to remain applicable. It is therefore necessary to implement techniques allowing for calling the `save` method on the root of an object graph and combining insert and update operations in few SQL queries.
- Regarding object graph traversal, eager object loading is important. Future versions of the framework should include this feature to minimize the number of SQL statements required to obtain the entire object graph.
- During the execution of the benchmark in development mode, it became apparent that preconditions for description and table update checks provide a vast performance improvement. Therefore, after the completion of the benchmark suite means have been integrated into the framework that not only prevent updates of descriptions and table structures, but also the examination of their predecessors if it is not utterly necessary.

### 4.5 Summary

The presented benchmark results have shown that SqueakSave still has to be optimized for certain fields of application. Especially the query performance for large result sets is an issue that deserves closer attention in the future. However, object graph traversals are implemented in a viable manner and the results demonstrate that the minimalist intrusion into object models has a positive impact on such operations. Additionally, the declarative nature of the query interface, as well as the simple set-up and integration of the framework are advantages that make SqueakSave a suitable persistence solution for application development in Squeak.

## 5. Related Work

The special capabilities of dynamically-typed object-oriented programming environments like Squeak or other Smalltalk dialects affect the design and implementation of O/R mapping solutions. While the possibility to analyze the source code before program execution to determine the required table structure is missing, the often much more elaborate introspection and intercession features allow for more flexible implementations. Within the scope of this paper, only mappers for dynamically-typed object-oriented environments are considered. However, since the mapper is to provide persistence in a manner reminiscent of object-oriented databases, examples of this category have also been investigated with regards to their support for a relational database foundation.

***Dynamic Object-Relational Mappers*** ActiveRecord for Ruby on Rails [12] is a database schema-driven O/R mapping solution that adheres to the *convention over configuration* (CoC) principle [28]. While it provides almost effortless configuration, database schemas and object models are not automatically kept synchronized. Especially alterations of the application object structure have to be manifested in the database schema before they are available in the respective object model and subject to persistence mechanisms. ActiveRecord also introduced dynamic finder methods as a language-native query interface for relational databases.

DataMapper[11], another Ruby O/R mapping framework, relies on mappings defined by a very minimalist API, that only requires the definition of an SQL type for a certain attribute in order to create a valid database schema. After each mapping change, a re-run of the database creation method has to be performed, but will consecutively erase the database completely and remove all data. However, the framework also offers migrations, that can gradually add, alter, or remove columns in existing database tables. The query API is quite similar to the one present in ActiveRecord.

GLORP [18] provides object-relational persistence by heavy utilization of meta descriptions. These must follow certain naming conventions and have to be declared for the model, the database tables, and the relation between model attributes and database constructs. While GLORP allows for comprehensive reverse mapping of legacy database structures, its addition to existing applications is impeded by the mandatory introduction of an *id* instance variable to each persisted model class, and the need to provide a complete mapping description even for trivial cases.

IOSPersistent[12] was following an approach similar to SqueakSave. It provided fully-automatic persistence for all subclasses of an abstract base class of the framework and automatically created the according table models. Due to its monolithic architecture, it was not extensible by simple means and additionally did not allow for custom object-relational mapping descriptions. It has been superseded by the ReServe[13] project, that removed the automatic table creation, but in contrast simplified the creation of custom mapping descriptions and introduced a query API, that has been the foundation for SqueakSave's language-native queries.

***Object Databases*** The Gemstone project [5] provides almost transparent persistence. However, it requires an extensive environment in order to be applied as a persistence solution. It generally relies on object-oriented database technology to persist application data, but additionally provides the means to integrate relational database management systems into the storage process.

Another object-oriented database that provides compatibility with relational systems is db4o [24]. The db4o Replication System (dRS) utilizes Hibernate to replicate application data to specified RDBMS and is additionally able to read data from relational databases. Thereby users are able to perform ad-hoc SQL queries on the data without having to utilize an environment capable of handling the db4o-internal data structures. Additionally, this feature allows the integration of legacy data from relational database into object-oriented environments.

## 6. Conclusions

SqueakSave is a reflective object-relational mapper that relieves developers of the task to manually maintain mappings between object models and relational database structures. Additionally, the framework is implemented in a way that does not interfere with existing object models and thus can be added almost transparently to existing solutions. While those features provide an increased degree of flexibility, query and storage performance are slightly diminished. However, since the main goal of the implementation has been to aid the development process of applications, the decreased performance is a trade-off that is worthwhile with regards to the gain in developer productivity.

The depicted extension points of the framework ought to support the development of new and innovative ways to create specialized table structures and mapping description formats that can be easily integrated into the existing solution.

While the current version is able to compete with long-established solutions, future work will especially involve the optimization of queries that deliver large data sets and the simultaneous insertion of multiple application objects within a decreased amount of SQL statements.

Another important aspect for improvement is the provision of custom mapping description handlers. Thereby, the seamless integration of SqueakSave into existing applications can be vastly simplified by enabling the framework to utilize descriptions that have already been created for other O/R mappers such as GLORP. Additionally, general purpose meta description frameworks, such as Magritte [27] could be integrated to not only map objects to relational constructs, but also generate validation methods that are performed before the storing of objects.

Despite the obvious optimization and extension points identified within this paper, other research projects could be adopted to further minimize the intrusiveness of the framework into the application or further optimize the generation of SQL queries. The former could be reached by utilizing aspect-oriented constructs to provide the persistence func-

[11] `http://www.datamapper.org`

[12] `http://www.squeaksource.com/IOSPersistent.html`

[13] `http://www.squeaksource.com/ReServe.html`

tionality as an easily attachable aspect to existing applications [26]. The latter is possible by an in-depth analysis of inner-application workflows, that determine the queries most suitable within certain execution states [25].

SqueakSave provides a solid foundation for further research and shows that meta-programming and reflection are viable to simplify the integration of object-relational persistence mechanisms into applications developed in dynamically-typed object-oriented programming environments.

## References

[1] S.W. Ambler. Designing a Robust Persistence Layer. *Softw. Dev.*, 6(2):73–75, 1998.

[2] S.W. Ambler. *Agile Database Techniques*. John Wiley & Sons, 2003.

[3] R. Barcia, G. Hambrick, K.Brown, R.Peterson, and K.S.Bhogal. *Persistence in the Enterprise*. IBM Press, 2008.

[4] A.P. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker. *Squeak by Example*. Institute of Computer Science and Applied Mathematics of the University of Bern, Switzerland, 2008.

[5] P. Butterworth, A. Otis, and J. Stein. The GemStone object database management system. *Commun. ACM*, 34(10):64–77, 1991.

[6] M.J. Carey, D.J. DeWitt, and J.F. Naughton. The 007 Benchmark. In *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 12–21, New York, NY, USA, 1993. ACM.

[7] M.J. Carey, D.J. DeWitt, J.F. Naughton, M. Asgarian, P. Brown, J.E. Gehrke, and D.N. Shah. The BUCKY object-relational benchmark. In *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 135–146, New York, NY, USA, 1997. ACM.

[8] W.R. Cook. Interfaces and specifications for the Smalltalk-80 collection classes. *SIGPLAN Not.*, 27(10):1–15, 1992.

[9] W.R. Cook and C. Rosenberger. Native Queries for Persistent Objects. *Computer Languages, Systems & Structures*, 31:127–141, 2005.

[10] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A.P. Black. Traits: A mechanism for fine-grained reuse. *ACM Trans. Program. Lang. Syst.*, 28(2):331–388, 2006.

[11] J. Elliott. *Hibernate: A Developer's Notebook*. O'Reilly Media, Inc., 2004.

[12] O. Fernandez. *The Rails Way*. Addison-Wesley, 2007.

[13] M. Fowler, D. Rice, M. Foemmel, E. Hieatt, R. Mee, and R. Stafford. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.

[14] E. Gamma, R. Helm, and J.M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[15] T. Goldschmidt, R. Reussner, and J. Winzen. A case study evaluation of maintainability and performance of persistency techniques. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 401–410, New York, NY, USA, 2008. ACM.

[16] J. J. Hallett and A. J Kfoury. A formal semantics for weak references. Technical report, Department of Computer Science, Boston University, 2005.

[17] E. Klimas, D. Thomas, and S. Skublics. *Smalltalk with style*. Prentice Hall, Englewood Cliffs, NJ, 1996.

[18] Alan Knight. GLORP: generic lightweight object-relational persistence. In *OOPSLA '00: Addendum to the 2000 proceedings of the conference on Object-oriented programming, systems, languages, and applications (Addendum)*, pages 173–174, New York, NY, USA, 2000. ACM.

[19] S.H. Lee, S.J. Kim, and W. Kim. The BORD Benchmark for Object-Relational Databases. In *DEXA '00: Proceedings of the 11th International Conference on Database and Expert Systems Applications*, pages 6–20, London, UK, 2000. Springer-Verlag.

[20] U. Leser and F. Naumann. *Informationsintegration: Architekturen und Methoden zur Integration verteilter und heterogener Datenquellen*. Dpunkt Verlag, 2007.

[21] F. Lodhi and M.A. Ghazali. Design of a simple and effective object-to-relational mapping technique. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 1445–1449, New York, NY, USA, 2007. ACM.

[22] S. Melnik, A. Adya, and P.A. Bernstein. Compiling mappings to bridge applications and databases. *ACM Trans. Database Syst.*, 33(4):1–50, 2008.

[23] OMG. UML 2.0 Specification, 2005.

[24] J. Paterson, S. Edlich, H. Hörning, and R. Hörning. *The Definitive Guide to db4o*. Apress, Berkely, CA, USA, 2006.

[25] P. Pohjalainen and J. Taina. Self-configuring object-to-relational mapping queries. In *PPPJ '08: Proceedings of the 6th international symposium on Principles and practice of programming in Java*, pages 53–59, New York, NY, USA, 2008. ACM.

[26] A. Rashid and R. Chitchyan. Persistence as an aspect. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 120–129, New York, NY, USA, 2003. ACM.

[27] L. Renggli. Magritte - Meta-Described Web Application Development. Master's thesis, Software Composition Group, University of Berne, 2006.

[28] C. Richardson. ORM in Dynamic Languages. *Queue*, 6(3):28–37, 2008.

[29] D. Thomas. Ubiquitous applications: embedded systems to mainframe. *Commun. ACM*, 38(10):112–114, 1995.

[30] D. Ungar and R.B. Smith. Self: The power of simplicity. *SIGPLAN Not.*, 22(12):227–242, 1987.

[31] P. Van Zyl, D.G. Kourie, and A. Boake. Comparing the performance of object databases and ORM tools. In *SAICSIT '06: Proceedings of the 2006 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries*, pages 1–11, Pretoria, Republic of South Africa, 2006.