Software Architecture Group
Hasso-Plattner-Institute
University of Potsdam

Master's Thesis

Squeak Save

An Automatic Object-Relational Mapping Framework

Thomas Kowark
April 28, 2009

Supervisors:
Dr. Michael Haupt
Prof. Dr. Robert Hirschfeld

Abstract

The maintenance of application data within persistent storage spaces is an important aspect of application architectures and development processes. Especially small development teams use dynamically-typed object-oriented programming languages for those development processes, since they provide a programming paradigm that embraces changing of existing implementations. Therefore, it is necessary that persistence mechanisms do not impede the agile application development, but instead can be seamlessly integrated into already present applications and also support their adaption to new project requirements. Persistence should, accordingly, be added as an almost transparent aspect that minimally intrudes the object model and the programming principles.

Many different solutions in the field of object persistence have evolved over the last decades and new approaches are still subject to research. One of the most popular techniques is the utilization of relational databases along with an object-relational mapping (O/R mapping) layer that translates object structures into relational constructs. While the required effort for the creation of mapping descriptions has been reduced drastically in current implementations that favor convention over configuration or utilize an extensive toolset for automatically generated mappings, the maintenance of object-relational mapping descriptions is still a task that impedes agile development processes.

In addition to the impact of this new description layer, other aspects of O/R mapping frameworks also influence the application development. Especially the means to perform queries on the underlying databases differ vastly in the degree of interface transparency. While some approaches implement query APIs that almost seamlessly integrate into the respective programming language, others offer only a minimalist abstraction from the corresponding database query language.

Another challenging task for O/R mapping solutions is the interoperability with other frameworks. While standard patterns for the creation of relational structures for object persistence exist, manual mappings or custom mapping conventions aggravate the joint usage of the same database by multiple O/R frameworks.

This thesis presents an approach to object-relational mapping that strives to eliminate the need for manually created mapping descriptions by the utilization of language-integrated features like meta-programming and reflection. The presented system uses those techniques to automatically derive suitable relational structures from the introspection of existing object models. Thus, it frees developers from the task to synchronize changes to object models and table structures and, accordingly, improves the required development times for the implementation of object model alterations. Additionally, the system provides a language-native query interface and means to support the collaboration with other O/R mapping solutions.

Zusammenfassung

Die Verwaltung von Anwedungsdaten in persistenten Speicherbereichen ist ein wichtiger Aspekt von Anwendungsarchitekturen und Entwicklungsprozessen. Besonders kleine Entwicklerteams benutzen dynamisch typisierte, objektorientierte Programmiersprachen für diese Prozesse, da sie ein Programmierparadigma bieten, das die Veränderung existierender Implementierungen erleichert. Deshalb ist es notwendig, dass Persitenzmechanismen die agile Entwicklung nicht behindern, sondern sich nahtlos in bestehende Anwendungen einbinden lassen und deren Anpassung an neue Projektvorgaben unterstützen. Persitenz sollte folglich als beinahe transparenter Aspekt hinzugefügt werden können, der das Objektmodell und die Programmierprinzipen nur minimal beeinflusst.

Viele verschieden Lösungen im Bereich der Objektpersitenz sind in den letzten Jahrezenten entstanden, und neue Ansätze werden immer noch erforscht. Eine der meistgenutzen Techniken ist die Benutzung relationaler Datenbanken im Zusammenspiel mit einer objekt-relationalen (O/R) Abbildungsschicht, die Objektstrukturen in relationale Konstrukte übersetzt. Obwohl der Aufwand für die Erstellung von Abbildungsbeschreibungen in neueren Implementierungen drastisch reduziert wurde, da diese Koventionen über Konfiguration stellen oder umfangreiche Werkzeuge zur automatischen Erzeugung nutzen, ist die Wartung von objekt-relationalen Abbildungsbeschreibungen noch immer eine Aufgabe die agile Entwicklungsprozesse behindert.

Zusätzlich zu den Auswirkungen dieser neuen Beschreibungsschicht beeinflussen auch andere Aspekte von O/R Ansätzen die Anwendungsentwicklung. Besonders die Methoden zur Ausführung von Anfragen an die zugrunde liegende Datenbank unterscheiden sich erheblich im Grad der Schnittstellentransparenz. Während einige Ansätze Anfrageschnittstellen bereitstellen, die sich nahtlos in die gewählte Programmiersprache einfügen, bieten andere nur eine minimale Abstraktion von der Datenbankanfragesprache.

Eine weitere Herausfordeung für O/R Lösungen ist die Interoperabilität mit anderen Ansätzen. Obwohl es Standardmuster für die Generierung relationaler Strukturen für Objektpersitenz gibt, erschweren manuell erstellte Abbildungen und indivduelle Abbildungskonventionen die gemeinsame Nutzung gleicher Datenbanken durch verschiedene Lösungen.

Diese Arbeit stellt einen Ansatz für objekt-relationale Abbildung vor, der versucht die Notwendigkeit manueller Abbildungserzeugung zu beseitigen, indem programmiersprachliche Eigenschaften wie Metaprogrammierung und Reflexion genutzt werden. Das vorgestellte System benutzt diese Techniken um automatisch geeignete relationale Strukturen aus der Untersuchung existierender Objektmodelle herzuleiten. Dadurch befreit es Entwickler von der Aufgabe, Änderungen von Objektmodellen und Tabellenstrukturen synchron zu halten. Folglich werden die Implementierungszeiträume für Änderungen am Objektmodell deutlich verkürzt. Zusätzlich bietet das System ein Anfrageschnittstelle, die in die Programmiersprache integriert ist, und Mittel zur Unterstützung der Zusammenarbeit mit anderen O/R Lösungen.

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Masterarbeit selbständig und ohne fremde Hilfe verfasst habe. Ich habe dazu keine weiteren als die angegebenen Hilfsmittel benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet.

Thomas Kowark Potsdam, den 31.03.2009

Acknowledgments

I would like to thank Prof. Dr. Robert Hirschfeld for the support and supervision of this thesis. Special thanks belong to Dr. Michael Haupt for many enlightening remarks regarding the implementation aspects of the thesis, as well as the extensive reviews of the written text. Furthermore, I am thankful to Arne Bergmann, Martin Beck, Michael Perscheid, and Bastian Steinert, who reviewed this thesis and helped me to improve its quality and contents.

Table of Contents

1 Introduction **1**

2 Background **5**

2.1 Persistence Management in Dynamically-Typed Object-Oriented Environments 5
2.2 Persistence Strategies 6
2.2.1 Relational Persistence 6
2.2.2 Object-Relational Persistence 7
2.2.3 Object Databases 8
2.3 Requirements for Automatic Object-Relational Support 9

3 SqueakSave **11**

3.1 Introduction 11
3.2 Basic Persistence Mechanisms 11
3.2.1 Initial Setup and Configuration 12
3.2.2 Persisting Objects 14
3.2.3 Object Query Interface 15
3.2.4 Multi-Developer Collaboration 16
3.3 Customization 18
3.3.1 Custom Configuration 18
3.3.2 Session Usage 19
3.3.3 Performance Optimization 21
3.3.4 Custom Object-Relational Mapping Descriptions 22
3.4 Summary 23

4 SqueakSave Framework Architecture **25**

4.1 Basics 25
4.1.1 Object-Relational Mappings - Creation and Update 25
4.1.2 Table Structure Adaption 29
4.1.3 Storage Wrapper Class 30

4.2 Utility Classes . . . 32
4.2.1 Object Caches . . . 32
4.2.2 Database Connection Handling . . . 33
4.2.3 Configuration and Customization . . . 33
4.3 Query Generation . . . 35
4.3.1 Collection Protocol Emulation . . . 35
4.3.2 Convention Based Query Methods . . . 36
4.4 Framework Extension . . . 37
4.4.1 Custom Object-Relational Mapping Descriptions . . . 37
4.4.2 Table Structures . . . 38
4.4.3 Database Adapters . . . 38
4.5 Summary . . . 39

5 Evaluation **41**
5.1 Performance . . . 41
5.1.1 Comparison with other Object-Relational Mappers . . . 42
5.1.2 Development vs. Production Environment . . . 47
5.1.3 Framework Profiling . . . 49
5.2 Interoperability . . . 50
5.3 Summary . . . 51

6 Related Work **53**
6.1 Static Object-Relational Mappers . . . 53
6.2 Dynamic Object-Relational Mappers . . . 54
6.3 Object Databases . . . 55

7 Summary and Outlook **57**

Bibliography **57**

List of Figures

3.1 Example Application Class Structure 12
3.2 Configuration Class Structure 13

4.1 Overview of SqueakSave System Classes 26
4.2 Workflow of Automatic OR-mapping description updates 28
4.3 Table Description Classes 29
4.4 Additions to the Object Protocol 30
4.5 Internal Workflow of the Save-Operation 31
4.6 Retrieval of Database Connections 34
4.7 Collection Protocol Emulation Classes 36
4.8 Description Handler Inheritance 38

5.1 OO7 Benchmark - Class Model 42
5.2 OO7 Benchmark - Database Creation Times for Different OR-Mappers 44
5.3 OO7 Benchmark - Query Times 45
5.4 OO7 Benchmark - Traversal Times 46
5.5 OO7 Benchmark - Database Creation Times for SqueakSave Modes 47
5.6 OO7 Benchmark - Query Times for SqueakSave Modes 48
5.7 OO7 Benchmark - Traversal Times for SqueakSave Modes 49
5.8 Example Application - Class Structure with ActiveRecord 51

Listings

3.1 Connection Specification. 12
3.2 Configuration Set-Up. 13
3.3 Basic Object Storage. 15
3.4 Query Examples - Emulated Collection Protocol. 15
3.5 Query Examples - Convention-Based Dynamic Finders. 16
3.6 Query Examples - Convention-Based Dynamic Finders on Classes. 17
3.7 Migration Usage. 18
3.8 Extended Configuration. 19
3.9 Alternative Ways to Retrieve Session Objects. 20
3.10 Transactions within Sessions. 20
3.11 Different Save Levels of SqueakSave. 21
3.12 Custom Mapping Description. 23

4.1 Language-Native Query Before Translation to SQL. 35
4.2 SQL WHERE Statement Generated from Language-Native Query. 35
4.3 Dynamic-Finder Method Before Conversion into Block-Closure. 37
4.4 Block-Closure Generated from Dynamic-Finder Method. 37

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 the scope of a project. While enterprise applications rely on robustness, execution speed and scalability [5], smaller projects additionally focus on the flexibility to quickly adopt to changes within the object model [4]. 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 the project scope, decisions regarding the development environment and language also influence the choice between available persistence strategies. Especially dynamically-typed languages vastly reduce turn-around and implementation timeframes by offering a programming paradigm that embraces change of existing implementations [52] and strong meta-programming and reflective features. Those techniques, however, impose non-trivial challenges for the implementation of persistence management systems. Since real-life applications exist that utilize features such as class creation and alteration during runtime [41], means have to be applied to persist instances of those classes as well.

Today many persistence strategies are available [10, 17, 33, 43, 49] to developers and their underlying data storage technologies cover a wide spectrum. It reaches from pure relational databases, through relational databases that have been enriched with object-oriented techniques, to completely object-oriented implementations. The ease-of-integration of those solutions into dynamic, object-oriented applications differs strongly [23], since the mismatch between the paradigms founding the application development and the persistence framework varies in its extent [4].

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 [3]. Even though it would be possible to implement a custom mapping layer for each application, the general rules of object-relational mapping follow a number of standard patterns and are therefore integrated into generic object-relational mapping frameworks. Those frameworks cover a variety of aspects reaching from basic CRUD[1] functionality to more elaborate features like transaction processing. However, most available systems require extensive meta-description of the object model in order to be able to perform the aforementioned tasks.

[1]Create, Read, Update, Delete

Such descriptions impose an undeniable burden on application development. Upon each change of the object model, the description layer has to be altered, as well [38, 40]. Additionally, manual mapping creation impedes the interoperability between different O/R mapping approaches. All design decisions that underly the mapping descriptions and do not follow standard conventions, have to be manually ported to the description systems of other O/R mappers whenever two applications are supposed to share a common data set.

Another important aspect regarding the seamless integration of O/R mapping frameworks into applications is the intrusiveness into the existing object and programming model. While overall a high degree of transparency of the underlying database structures and systems should be achieved [36], the 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 and the need to alter inheritance hierarchies or even object layouts in order to store objects within relational databases.

Contribution The objective of this thesis is to develop and evaluate a framework that automates the creation and maintenance of object-relational mappings. All changes to an application's object model are reflected by altered mapping descriptions, as well as adaptions of the resulting relational database schemas. By following numerous standard patterns for the transformation of object structures into database structures, the framework is supposed to provide a high degree of interoperability with other solutions.

As previously mentioned, non-intrusiveness into object and programming models is also essential for a transparent integration of the framework into applications that require object-relational persistence. Therefore, the framework's implementation is carefully designed to minimize the required alterations.

Additionally, extension points for existing functionality are a viable aspect of the implementation. By incorporating means to adopt different mapping patterns, database access techniques and mapping description flavors, the framework should not only be able to interoperate with existing solutions, but also with new frameworks that are yet to be developed.

This thesis therefore proposes an object-relational mapping framework using meta-programming and reflective capabilities to introspect applications during their runtime and automatically derive the mappings and table structures that are required to persist objects in an almost transparent manner. The framework is named SqueakSave. Due to its highly incisive object metaphor, that is manifested not only within the language itself, but also in the tools available within the programming environment, Smalltalk [22] has been selected as the dynamic programming language for the implementation of the framework.

Structure of the Thesis In order to facilitate a profound understanding of the underlying technologies, chapter 2 provides insights into the distinctive challenges of persistence in dynamic, object-oriented environments. Additionally, different approaches to the subject of object persistence are presented and evaluated with regards to their suitability for the previously mentioned tasks. The chapter concludes with an analysis of the requirements for an object-relational persistence framework and thus provide the foundation for the description of the framework itself.

Chapter 3 presents the intended usage workflow for the integration of SqueakSave into applications as the chosen persistence management system. This includes a detailed description of the basic mechanisms for simple, straightforward implementations, as well as more elaborated constructs like custom object-relational mapping descriptions. A running example accompanies this chapter.

Following the usage description, the architecture of the framework is discussed in chapter 4. While the focus resides on how the mechanisms offered by the system have been implemented, topics like thread safety and performance improvements through the usage of object caches are also reviewed. The second part of this chapter describes essential extension points for various parts of the framework.

Evaluation of the implementation is the topic of chapter 5. The SqueakSave implementation of the aforementioned guiding example is compared to an implementation that uses a different O/R mapper. Additionally, profound performance measurements are presented. This includes a standard benchmark, which is performed with multiple persistence frameworks, as well as a detailed analysis of the different operational modes provided by SqueakSave.

Related work in the field of object persistence technologies is dealt with in chapter 6. A distinction between techniques for dynamic object-relational mappers, as well as for those dealing with statically typed languages is made. Additionally, some interesting solutions in the field of object databases are presented, as they strive to provide compatibility with relational systems.

The thesis is summarized in chapter 7 along with a brief discussion of possible future work, which can be based on the introduced implementation.

2 Background

Various different strategies exist to enrich applications with the means to persist objects. In addition to the strategy descriptions the development environment used for the implementation is presented. The chapter concludes with a discussion of the requirements that can be derived from the analysis of other persistence frameworks and the peculiarities of the chosen development environment.

2.1 Persistence Management in Dynamically-Typed Object-Oriented Environments

Dynamically-typed object-oriented environments imply numerous challenges for the implementation of persistence management systems. The most obvious one is a direct consequence of the dynamic typing of objects: It is possible to assign each variable within the system of arbitrary types without altering variable definitions or method signatures.

While this behavior aids agile development of applications [39], it also enforces persistence strategies to be highly flexible in order to not impede the development process. If for example relational persistence is the technology of choice, changing the type of an instance variable also has to be taken into account for the database schema required to store instances of a particular class.

The possibilities provided by meta-programming and reflection capabilities are another important aspect of the desired programming environment. Especially the creation of new classes as well as the alteration of existing ones during the execution of an application are very challenging tasks for a persistence framework [41]. Since no user-written database mapping can be presumed for such classes, the framework itself has to provide means to serialize this data into matching database structures. However, since persistence frameworks for a particular programming environment are usually being written within the language itself, the same reflective and introspective capabilities can be used in order to deduce the corresponding database schemas.

In addition to the intended changes of class structures, dynamically-typed environments also are more error-prone with regards to unintentionally assigning values of wrong types to instance variables of objects. Small semantic errors can lead to objects of the same class with substantially different inner-object structures. A persistence framework for such an environment has to take this possibilities into account and accordingly provide means to inform developers about ambiguous object structures that could lead to serious loss of data.

2.2 Persistence Strategies

The most simplistic way of persisting objects within dynamically-typed object-oriented environments is to use the language-integrated means to serialize them into a string representation and store those strings within the file system. In order to import this serialized objects into other environments, standardized serialization languages such as YAML[1] can be used if the corresponding parsers and writers are available in both languages. Image-based solutions such as most Smalltalk environments, additionally offer the possibility to create simple collections of objects, which will remain within the image itself, for as long as the collection is being referenced.

While those approaches might be quite viable for the initial phase of a project, they lack important features, that will be of essence when projects advance or multiple developers become involved. Amongst others, this includes the ability to execute queries on the existing data in a scalable manner and perform persistence operations within transactions [4].

2.2.1 Relational Persistence

Relational Database Management Systems (RDBMS) allow for creating, reading, updating and administering relational databases. Due to the proven mathematical foundation of their underlying relation model [13] and the availability of the technique for more than 30 years, various differing implementation flavors arose. Amongst closed-source projects, Oracle databases are the most popular solution[2]. In the field of open-source databases MySQL is the market leader followed by PostgreSQL according to a global survey[3].

The most important asset of RDBMS is their support for transactional behavior [5]. This allows for altering data according to the ACID paradigm, that requires database operations to be `a`tomic, `c`onsistent, `i`solated and `d`urable. By adhering to this principle, data integrity can be guaranteed, since multi-step operations will either completely succeed, or the database is rolled back to the state it has been in before the beginning of the transaction. Additionally, checks will be carried out to determine that referential integrity is maintained; thus, no reference is pointing to non-existing or unsuitable data.

Querying the data repository is possible by using the structured query language (SQL) [28]. Since it is standardized by the International Organization for Standardization (ISO), queries developed for one particular RDBMS can be transferred to another RDBMS, if both systems adhere to the standard specification. However, most vendors introduced proprietary extensions to their respective implementations [24], a fact that leads to decreased portability, if the users are unaware of the standard violations. The queries itself are subject to mathematics-based optimizations carried out by the DBMS. While each system might utilize its own optimization engine, the general principles of query-optimization have been subject to research since SQL has been introduced [29].

[1] `http://www.yaml.org`
[2] `http://www.gartner.com/it/page.jsp?id=507466` (03/03/2009)
[3] `http://www.alfresco.com/community/barometer/files/wp-osb-III.pdf` (03/23/2009)

2.2.2 Object-Relational Persistence

Features like optimized queries, transaction support, or portability of schemas and queries qualify relational database systems as reasonable choices for data persistence [5]. However, their applicability for the task of persisting objects, created in object-oriented applications, is degraded by the so-called impedance mismatch [4].

The relational and the object-oriented paradigm differ in various aspects. Most obviously, the data types used within the chosen object-oriented programming language do not necessarily comply with the data types available in the respective RDBMS. While this can be solved in a very straight-forward manner for simple data types such as strings or integers [4], collections have no direct and unique representation in a relational database. A possible solution is to add a foreign key column to the table representing the objects within the collection. This key references the object that owns the collection. Another valid approach would be to use a join table that establishes an association between the unique id of the collection owner and each id of the collection objects.

Inheritance is another aspect that can be mapped to relational models in an ambiguous manner. Three techniques have been proposed to map inheritance structures to relation table structures - class, single and concrete table inheritance [20]. However, in addition to those techniques, other approaches are also possible, such as providing an extra table that only stores the type of an object within a single column.

Object-Relational Database Management Systems In order to overcome these ambiguities, relational database systems have been enriched with constructs that allow for a straight-forward mapping of most object-oriented techniques to relational constructs [50]. Those object-relational database management systems (ORDBMS) offer means like table inheritance or user-defined data types and collections, that enable developers to facilitate them as a direct representation of the structures existing within their application. By utilizing table inheritance, for example, each table can directly represent a class and the super and sub-class relationships will be depicted by defining super and sub-tables respectively.

However, while those features can provide a simple representation of object-oriented structures within enriched relational databases, their usage still remains SQL-based. This requires developers to incorporate SQL statements into their application code and accordingly will complicate refactoring processes. Since the queries are mostly represented by strings, each of those has to be evaluated for references to actual object properties and their database representations, e.g. column names. While this could be carried out automatically by integrated development environments (IDEs), most of them do not introspect embedded strings for semantic or syntactic errors and also do not include them in refactoring measures [19]. Therefore it is always advisable to incorporate a persistence abstraction layer into the application, that encapsulates all SQL related operations in a single point of access [55].

Object-Relational Mappers In addition to the mere encapsulation of SQL statements within a separate architecture layer, object-relational mappers (O/R Mappers) provide means to describe the relationship between object-oriented constructs and relational database schemas [4].

This meta-descriptions of the objectmodel can be either explicitly manifested in description objects, XML documents, or source-code annotations, or be implicitly inferred through naming conventions [49]. Thus, the foundation of the description does not necessarily has to be the object model itself, but can also be an existing legacy database schema [55].

By utilizing this meta-data, it is possible for object-relational mapping frameworks to provide basic CRUD (Create, Read, Update, Delete) functionality for objects that are persisted within a relational database. The mapping thereby defines which columns and tables have to be queried or altered, when the respective attribute values of objects within the application change.

While this approach is today widely acknowledged as a valid technique to diminish the efforts required in order to overcome the previously mentioned impedance-mismatch, its implementations greatly differ in terms of performance and maintainability [23]. Especially frameworks that, in addition to the already process-inherent data modeling techniques, require the maintenance of user-generated mapping descriptions enforce a new modeling level, that has to remain synchronized with the object model of the application [38]. Thus, a high degree of flexibility regarding the creation of meta-data is essential for an O/R Mapper in order to not only solve the object-relation impedance mismatch, but also to aid the development process of applications.

Another aspect that has to be covered by O/R mapping frameworks is the creation of a generic, general-purpose query API. Since the mapping description between objects and relation constructs is already present, the query processing engine has to take advantage of this prior knowledge, and should not lead to an implicit duplication of mapping aspects. This might happen if the query engine incorporates strings that only depict the database representation of an attribute instead of utilizing the attribute directly [15]. Therefore, the query engine should be built upon language constructs of the chosen application environment instead of forcing developers to utilize another indirection layer with a different syntactical structure.

The huge variety of available O/R mapping approaches leads to the problem of interoperability. While, generally, full-fledged O/R mappers should be capable of mapping arbitrary database schemas to object structures and vice versa, an important issue is the development effort required to port an application from the usage of one mapper to another. Generic schemas as described in [20] should therefore always be the standard target of O/R Mappers, and should only be altered by user request. If, however, multiple applications are supposed to share the same database, then each of the mappers, if not capable of producing a compliant schema, should be configurable in a merely trivial way to adopt to the specific peculiarities of different frameworks. This, amongst other fields, includes the possibility to swiftly change from one inheritance mapping to another or to specify a standard pattern for the mapping of one-to-many associations to the respective database tables.

2.2.3 Object Databases

The most natural way to persist objects created within an object-oriented programming environment is to utilize object-oriented database management systems (ODBMS) [31]. Data is stored in a manner that is close to the internal representation of objects within the respective environment but is enriched with means to index data attributes in a way that allows for fast querying. A study [56] has shown that for most common use-cases the performance of object databases is far beyond the capabilities of relational database systems coupled to an object-oriented application

through an object-relational mapping framework.

Despite this advantage in query, storage, and adoption times, object databases suffer from some major drawbacks that diminish their relevance in application development. The most important aspect is the lack of adoption of the standard for object databases [34]. While the ODMG 3.0 standard has been published in 2001 [6], and also proposed the object query language (OQL) [2] as a standard for performing queries within object databases, neither have been completely adopted by a vast majority of object database vendors [34]. This reduces the interoperability between different ODBMS solutions and thus complicates the joint usage and migration of persistent data between multiple heterogenous applications.

2.3 Requirements for Automatic Object-Relational Support

Since the success of ODBMS seems not to be circumvented by the lack of a technological advantage, but due to economic considerations, object-relational technology will supposedly remain to be of importance within application development. Therefore means have to be established that simplify their usage, flexibility and interoperability with other solutions.

As stated in section 2.2.2 the synchronization between an application's object model and the mapping descriptions required to persist the created objects within relational databases is an issue that needs to be solved in a user-friendly manner. Various O/R mapping frameworks have been developed with the intention to minimize the effort of this maintenance process by either simplifying the description API, relying on naming conventions for object and database attributes, or developing tools for static source code analysis. Based on an examination of existing approaches like ActiveRecord for Ruby on Rails [18], the generic lightweight object-relational persistence framework (GLORP) for Squeak [33], or the Grails Object-Relational Mapper (GORM)[4] for Groovy, the following requirements have been deduced for an O/R mapper implementation:

- Simple and optimally automatic maintenance of object-relational mapping descriptions,
- Compatibility with other O/R mapping solutions with regards to created database schemas and mapping descriptions,
- Minimal interference with existing object models by rejecting the necessity to add inheritance relationships and instance variables that are only required for persistence purposes,
- A language-native query interface that is not based on string values, but on real method invocation, and thus allows for simplified refactoring of applications,
- Support for multi-developer collaboration in terms of consistent data repositories, database schemas and object-relational mapping descriptions,
- An implementation that in addition to flexibility, ease-of-use, and compatibility remains competitive with regards to performance of data manipulation and querying.

[4] http://www.grails.org/GORM

Especially the automatic maintenance of O/R mapping descriptions and the according database schemas is a feature that is still subject to research [38, 40] and is mostly implemented by the utilization of external tools instead of language integrated mechanisms.

This thesis therefore proposes an approach to object-relational mapping that makes capacious use of meta-programming and reflection features in order to automatically create and maintain O/R mapping descriptions during the runtime of an application. The solution implements the aforementioned requirements by using only the capabilities of its programming environment and renounces the need to utilize external tools, alter object and inheritance structures, or create new modeling layers for application developers.

Development Environment Squeak[5], an open-source Smalltalk dialect, has been chosen as the development environment for the implementation of the proposed object-relational mapping framework. It has been developed with a focus on educational purposes [27] and is available on a variety of platforms. The strong adherence to object-oriented principles and the resulting powerful meta-programming and reflective capabilities provide a supportive platform for the development of a system that is intended to rely heavily on those techniques. Additionally, available tools, such as a debugger featuring code replacement during application run time, a powerful refactoring engine, and a unit testing framework, provide a solid foundation for the implementation.

While the meta-programming facilities aid the development of the framework, another feature of Squeak imposes some challenges to the implementation. The image-based nature of the environment has to be regarded with respect to the possibilities it provides for the implementation of a shared object space between all processes running within that image. This could improve the memory footprint of the proposed O/R mapping solution, but close attention has to be paid to synchronization and thread-safety issues.

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

3 SqueakSave

The following chapter will provide an introduction into the usage patterns of the SqueakSave O/R mapper. A guiding example is used in order to simplify understanding of the basic concepts of the framework, as well as the more advanced features presented in section 3.3. While those descriptions show the intended usage, technical details of the inner workings will be presented in chapter 4.

3.1 Introduction

The example that accompanies this chapter has been chosen with regards to simplicity as well as recognizability. A weblog is an example that complies with both criterions. It is often utilized in introduction demonstrations of web frameworks and their respective standard persistence mechanisms[1], and also rather simple, and thus easily comprehensible, in its class structure.

The class structure of the sample application, which is depicted in the UML [42] class diagram in figure 3.1, includes the most common scenarios that object-relational mappers have to handle within applications [20]. The `User` class and its subclasses require the mapping of class inheritance to database structures. Additionally, in order to persist the `accountData` attribute, one-to-one (1:1) relations have to be implemented. The `blogs` property depicts a many-to-many (m:n) relation, since each blog can be maintained by multiple users. Finally, a one-to-many (1:n) relation has to be modeled in order to connect a blog object to its respective blog entries. In addition to the mapping of those relations, the chapter also depicts means provided by SqueakSave to customize the standard, auto-generated mappings, declare transient attributes, that should not be stored within the database and to improve compatibility with other object-relational mapping solutions, such as ActiveRecord for Ruby on Rails.

3.2 Basic Persistence Mechanisms

A main requirement for SqueakSave is to provide straightforward persistence mechanisms in a very simple manner. The following subsections will 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, query for objects based on certain attributes, and support the collaboration between multiple developers.

[1] e.g. `http://rubyonrails.org/screencasts` or `http://onsmalltalk.com/`

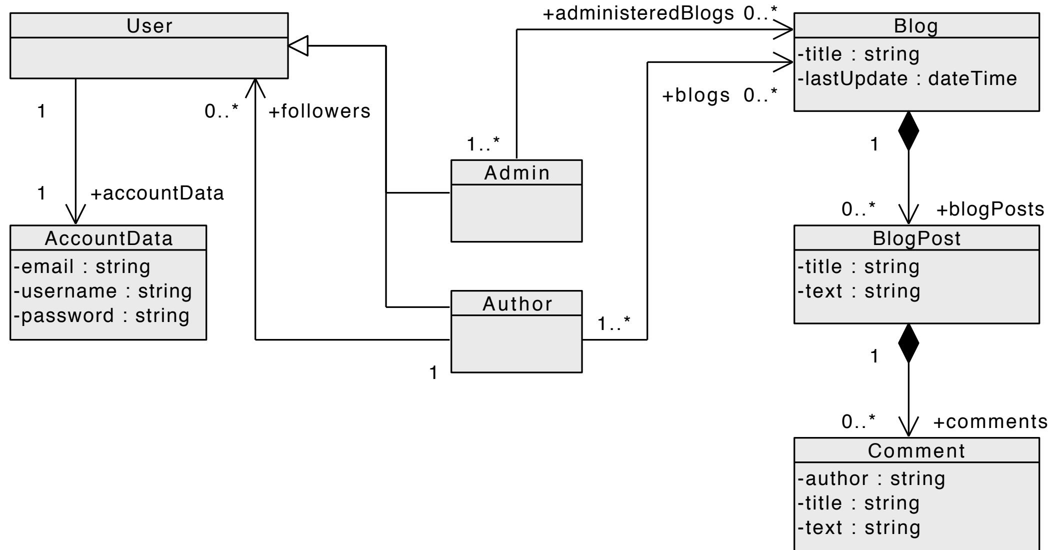


Figure 3.1: Class Structure of the Example Application.

```
SqsMySQLConnectionSpecification user: 'admin' password: 'password' database: 'blog_example_db'
```

Listing 3.1: Connection Specification.

3.2.1 Initial Setup and Configuration

For each class of objects that need to be persisted, developers have to set-up an instance of `SqsConfiguration`. As figure 3.2 shows, a configuration object has numerous properties that determine the behavior of the framework for the classes it applies to. Within the straightforward case, however, only the `connectionSpecification` attribute is important. The remaining properties will be discussed in section 3.3.

The connection specification determines which RDBMS is used as the target storage for the respective objects. For each supported system, the framework includes a specialized subclass of `SqsConnectionSpecification`. It provides standard values for port and hostname of common RDMBS server implementations such as MySQL or PostgreSQL. The only mandatory information are username, password and the name of the target database. Listing 3.1 depicts a minimalist connection specification for the example application. It is important that the user account specified for accessing the database has the privileges to create, alter, and drop tables, since SqueakSave is constantly reorganizing the table structure according to changes within the application classes.

All other properties of the created `SqsConfiguration` instance will be prefilled with standard values and, accordingly, only specifying the connection specification will always create valid configurations.

In order to register a configuration for the application classes, it is necessary to create a sub-

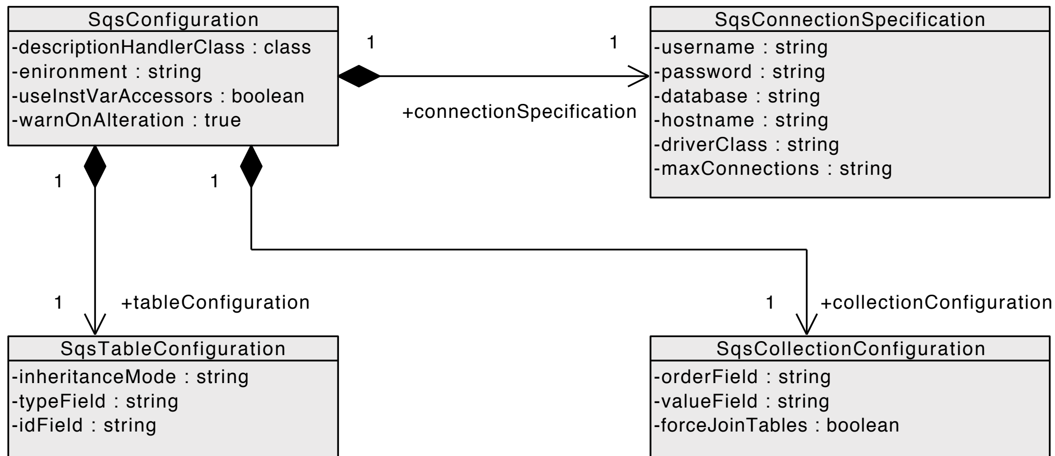


Figure 3.2: Configuration Class Structure.

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

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

Listing 3.2: Configuration Set-Up.

class of `SqsConfig`. The name of this subclass has to follow certain naming conventions to be recognized by the framework as being valid for classes of objects that ought to be subject to persistence operations. To create a configuration for the entire application, the first part of the class category, which is normally subdivided by '-' characters [9], has to be the first part of the class name followed by the suffix *SqsConfig*. The class-side method `connectionSpecification` has to be implemented to return valid server access credentials, according to the previously described constraints. The configuration class for the example configuration is depicted in listing 3.2.

Following those naming conventions, it is possible to create differing 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 (see section 3.3), it is possible to reimplement the `configuration` method on the class side of the configuration or application class, respectively. 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 [32] to directly connect the class category with its configuration, this is not possible within Squeak, since the category is only identified as a string and not accessible as a first class object.

3.2.2 Persisting Objects

Convention-based setup of configuration classes is essential to enable simple storing of objects. By patching the `Object` class, methods have been introduced that implement the data-modifying CRUD [55] operations: creating, updating, and deleting objects. As a consequence of this so-called 'monkey-patching'[2] each 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. Due to this implementation technique the framework does not force its users to alter the inheritance structure of their application to persist objects with SqueakSave. The drawbacks of renouncing the need to use an abstract base class will be discussed in section 4.1.

Listing 3.3 presents the creation of a user object with the according account data attribute. The `save` method will store the author object itself and the account data 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 that correspond to an object and additionally remove all references from other database tables as well. Accordingly, removing a user object from 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.

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

```
accountData := AccountData new
   password: 'password';
   username: 'testuser';
   email: 'user@example.org'.

author := Author new
   accountData: accountData.

author save.
```

Listing 3.3: Basic Object Storage.

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

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

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

Listing 3.4: Query Examples - Emulated Collection Protocol.

3.2.3 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 build upon a relational database foundation, those queries have to be carried out as SQL statements. As mentioned in chapter 2.2.1, integrating queries in such a way, that standard language constructs can be used, is an important feature with regards to the usability of an object-relational mapper [15]. Due to that fact, SqueakSave provides a query interface that does not rely on string-based query encoding, but instead emulates the Smalltalk collection protocol [14].

The standard target for object queries are instances of `SqsSearch`. They have to be instantiated with the class of objects, the search is supposed to return. While queries can be performed on each class residing within an image, a prerequisite is the availability of a valid configuration for at least the category, the class belongs to. The query will return instances of the query class itself, as well as of all its subclasses. 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 3.4 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 at least one blog with more than 10 blog posts. It has to be noted that this query would not work on the general User class, since the blogs attribute is only defined for instances of Author.

The last query is useful to determine 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 message-sends to the query objects, such as aBlogPost 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 have to be implemented within the SqueakSave framework. In chapter 4.3 we will present the implementation details of the query processing and depict means to extend the available protocols.

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 [49] for Grails[3] or ActiveRecord for Ruby on Rails.

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

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

Listing 3.5: Query Examples - Convention-Based Dynamic Finders.

The first query presented in listing 3.5 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. The keyword 'and' implies that the author and the title attribute have to match the specified arguments. Keywords 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 can 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 [16]. They have been introduced in the Self programming language, [53] and later been applied to Squeak, to provide a more fine-grained mechanism for reusing existing implementation details.

By adding the TSqsSearch trait to any class of an application's object model, queries can be performed like depicted in listing 3.6.

3.2.4 Multi-Developer Collaboration

An important aspect of application development is the collaboration between multiple developers [39]. Especially when the object model is constantly altered by more than one developer, the

[3] http://www.grails.org/

```
Blog findByTitle: 'testblog'

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

Listing 3.6: Query Examples - Convention-Based Dynamic Finders on Classes.

resulting changes of the underlying database schema have to be made persistent and thus subject to version control.

SqueakSave relies on class meta-descriptions in order to be able to create according table structures or perform search queries on the data. Those meta-data are continuously updated by the framework and thus reflect the changes that have to be carried out within the database. While only changes to the object model will be sufficient to keep the descriptions synchronized between multiple developers, as they will eventually lead to updates of the meta-data, it is not guaranteed that execution paths leading to an update of the table structure will be executed. Therefore, the framework is storing each newly created or updated description in the format specified by the chosen description handler (see section 3.3.4).

Those artifacts, which can be methods, XML files or code annotations, can be subject to version control systems, such as Monticello [37] or Subversion [45]. Upon each repository checkout the local versions will be replaced with the descriptions that other developers have recently created. Hence, they will remain consistent between all development environments.

Another aspect covered by SqueakSave is the creation of a common data foundation for all developers. This aspect is important for the development process, as it allows to have a well-defined set of data for testing changes to the application. Additionally, viable records, like shared geo databases, can be declared and altered by this means in order to make them available to the entire development team.

The class `SqsMigration` provides a mechanism that is quite similar to the migration technique introduced in Ruby on Rails [18].

Listing 3.7 presents the required steps to create migrations for the sample application. The migration class, which has to inherit from `SqsMigration`, can be used to automatically generate migration methods. This step will also inject a version pragma into each method, which defines an internal order for all migration methods. Accordingly, the methods will be performed in that particular order upon the invocation of `run` on the `BlogExampleMigration` class. The pragmas also guarantee that no migration is called for a second time after it has been successfully executed.

Within the migration methods, developers are free to create, load and alter objects as needed for their application development process. Each migration will be executed within a transaction. Accordingly, errors within the method code will not lead to inconsistent database states. Therefore, migrations provide a means to create a common and consistent database schema and content for all involved developers. A reasonable application would be the creation of a `BlogExampleTestMigration` that creates the aforementioned test data set and is performed before each run of unit tests.

To revert the changes to the database schema or content, the method `runDown` can be triggered.

```
SqsMigration subclass: #BlogExampleMigration
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'BlogExample'.

BlogExampleMigration generate: #createTestUser.

BlogExampleMigration>>#createTestUserUp
   <version: '20090321142023'>
   testUser := User new.
   user accountData: (AccountData new username: 'testuser'; password: 'password').
   user save.

BlogExampleMigration>>#createTestUserDown
   <version: '20090321142023'>
   testUser := User detect: [:aUser | aUser accountData username = 'testuser'].
   testUser destroy.

BlogExampleMigration run.
```

Listing 3.7: Migration Usage.

It will perform all methods with the suffix 'Down' and, for example, destroy the previously created test user.

3.3 Customization

Utilization of the presented techniques to store and query for objects as well as to aid multi-developer collaboration, is sufficient to perform basic CRUD operations on application data. However, as depicted during the requirements analysis in chapter 2, those basic features have to be extended with means to customize the behavior of the object-relational mapping framework and optimize aspects of performance and robustness.

3.3.1 Custom Configuration

The configuration object depicted in figure 3.2 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. By altering those values, it is possible to achieve a higher degree of interoperability with other ORM solutions.

Listing 3.8 presents the custom configuration method, which is required to customize the framework's behavior and standard values for the entire application category (depicted here) or for single classes and sub categories (see section 3.2.1). The attributes that refer to field names can be altered not only to adhere to naming conventions of other O/R mappers, but also to solve naming conflicts. An instance variable named *type*, for example, would be mapped by

```
BlogExampleSqsConfig class>>#configuration
   config := super configuration.

   config
      warnOnAlteration: false;
      useInstVarAccessor: false;
      environemnt: #production;
      descriptionHandlerClass: SqsPragmaDescriptionHandler.

   config tableConfiguration
      inheritanceMode: #classTableInheritance;
      idField: 'oid';
      typeField: 'class_name'.

   config collectionConfiguration
      orderField: 'index_value';
      valueField: 'collection_element'.

   ↑ config
```

Listing 3.8: Extended Configuration.

the framework to a column of the same name. However, since the default value of the column, that stores the class name of an object, is also *type*, the standard value should be altered.

In addition to those interoperability issues, altering the configuration can be used to define the behavior of the framework during runtime. Therefore it is possible to define whether instance variable accessor methods should be used or if access to the variables should be implemented by directly accessing their values. This can be especially important if developers do not want to implement those methods, or if they implement lazy initialization of objects [8].

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 those warning dialogs.

At some point during the development process, the object models might be complete, and changes are no longer likely to happen. When this degree of completeness has been reached, the introspection and mapping update functionality are no longer required, and 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'.

3.3.2 Session Usage

While the implementation of SqueakSave frees users from the need to utilize an explicit session object to store, retrieve and delete persisted 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`, that caches the sessions on a per-thread basis. Thus, requesting a session for a certain configuration, class, or category will always return the same

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

Listing 3.9: Alternative Ways to Retrieve Session Objects.

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

session inTransactionDo: transactionalBlock ifError: [ testuser rollback ].

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

Listing 3.10: Transactions within Sessions.

object within a single thread of control. The different possibilities to get the current session for the sample application are depicted in listing 3.9. The first two methods will deliver the same thread-specific session object, that contains the previously set-up standard configuration. The last possibility will create a new session, because of the custom configuration.

With the retrieved 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[4] 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. Therefore, the transaction does 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 it via offered functionality of the session protocol. However, this utilization can be error-prone, since viable control-paths, whose execution is guaranteed, have to be determined.

Listing 3.10 depicts the two possibilities by using an explicit session object, that has been retrieved like shown in listing 3.9. The `rollback` method, that is available for all objects just like the methods mentioned in section 3.2.2, will set the instance variables of the user object back to the pre-transaction state.

Since a session object also contains a copy of the respective configuration, the behavior of the framework can be explicitly defined for each session. A field of application of this technique are administrator tasks that, in contrast to the common behavior of a web page, rely on the pos-

[4] `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 blogs add: newBlog.

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

Listing 3.11: Different Save Levels of SqueakSave.

sibilities to alter data structures or inspect the generated SQL statements for certain operations. So basically, by overriding the configuration of the database session after an administrator has logged in to the application, it is possible to define a role-based behavior of the SqueakSave framework.

3.3.3 Performance Optimization

The database schemas created by SqueakSave follow the basic patterns described by Fowler et. al [20]. 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).

SqueakSave provides simple means to alter the generated table structure in a way that is more suitable for the requirements of individual applications. Abstract base classes can be declared by implementing the `sqsAbstractBaseClass` class method in the respective model class. Subsequently the class will be ignored with regards to table creation, and all attributes defined within this class will be mapped to the tables created for each of its direct subclasses. While the innerworkings of this behavior should not be of interest to framework users, it provides a simple starting point to overcome performance problems.

The table structure can be altered by setting the `inheritanceMode` value like depicted in listing 3.8. Since those configurations can be set for each class individually, it is possible to use different inheritance models for each application class. However, it is not possible to alter the table structure within one inheritance chain.

Another important aspect regarding the performance of the framework is the declaration of transient, i.e., not persisted, instance variables. If, for example, the user objects within the sam-

ple application would include instance variables that are only set during a visit of the web site as temporary storages, it would not be necessary to store them within the database. Those variables can be defined by implementing a class-side method named `sqsUnpersistedInstVars` on the respective class, which returns a collection of their names.

In addition to simple alterations of the created table structures, SqueakSave offers means to control the object graph traversal depth required to store or update objects. In addition to the already mentioned `save` method, developers are able to utilize `flatSave`, `deepSave`, and `saveToLevel:aLevel`. Within the example, that is presented in listing 3.11, the consecutive usage of those methods will gradually store more associated objects of the user.

After the `flatSave` command, only the user itself will be stored, and all newly created object are ignored. `Save`, as insinuated earlier, stores the new blog object in the database, but will not store the according blog posts or comment. The method that allows for the highest degree of control is `saveToLevel:aLevel`. With the specified level of two, the framework will follow a maximum of two references from the user object to find objects that need to be persisted. Hence, the new comment will not be stored. In order to persist the entire object graph, `deepSave` should be utilized. However, while this method guarantees that all reachable objects will be stored, it can have a considerable impact on the performance. Developers should therefore carefully decide which method to use, when certain storage operations exceed the expected runtimes.

3.3.4 Custom Object-Relational Mapping Descriptions

The customization of auto-generated mapping descriptions can become an important feature, when naming conventions of the O/R mapper collide with the expectations of developers, or whenever legacy data has to be mapped to an object model.

As stated in section 3.2.4, the mapping descriptions are not only kept in memory, but are also persisted. The format of this persistence is defined by the chosen description handler class. This can be altered within the configuration object itself (see listing 3.8). The standard description handlers utilize the internal format of the meta-descriptions. However, custom mapping descriptions, such as pragmas or XML documents can be generated, as well, if the corresponding description handler class has 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.

As far as the standard description handler is concerned, each of the description objects includes a `manuallyMaintained`-flag that indicates whether it is maintained by users or not. If this flag is set to true, the automatic updates will not alter the particular description. However, if the custom description requires changes to the database schema, those 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 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. This alteration does not require the alteration of the mapping description, but the creation of a class side method named `sqsTableName`. This method can be implemented within each class of an inheritance chain, without affecting the name mappings for super- or subclasses. Listing 3.12 depicts a custom configuration for the *username*

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

Listing 3.12: Custom Mapping Description.

field of the account data. The name of the column is changed to *name* and, additionally, the SQL type of the column is manually set to VARCHAR(20) to disable the automatic string size retrieval of SqueakSave (see chapter 4.1.1).

3.4 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 [26] and within this layer the presented CRUD-functionality can be implemented in a very intuitive manner.

A more detailed evaluation regarding the interoperability between SqueakSave and other object-relation mapping frameworks, will be presented in chapter 5.2.

4 SqueakSave Framework Architecture

In order to fulfill the requirements of flexibility and ease-of-use, SqueakSave's architecture has to incorporate means to free users from manually creating meta descriptions but still enable them to alter aspects of its inner workings. This requires to not rely on hard coded values, but to allow for dynamic configuration during application runtime. The following sections depict the implementation architecture of the basic classes that are required to provide simple CRUD functionality, as well as aspects concerning the extensibility of the framework.

4.1 Basics

The usage workflow described in chapter 3.2 is realized by the core classes of the SqueakSave framework. They are depicted in a simplified manner in figure 4.1, i.e., without the inclusion of concrete subclass implementations. In addition to handling the automatic description updates and CRUD operations those classes also have been implemented with a focus on object caching and thread safety.

The Petri net [44] depicted in figure 4.5 presents the basic workflow of persisting an object, either by inserting it into the database or updating the corresponding database rows. It becomes apparent that multiple actors of the framework are involved in this process, and accordingly the customization aspects have to be reflected within all those classes.

4.1.1 Object-Relational Mappings - Creation and Update

The mapping of class hierarchies and object attributes to relational constructs is performed by an introspection mechanism. Each object that is subject to persistence operations is examined and the according relational constructs are described according to a set of general mapping rules. As presented in chapter 3.3 the general rules can be overwritten with custom values.

General Mapping Rules For most basic data types, such as strings or integers, the mapping to relational constructs is straightforward. For each of the attributes that only holds a value of such a type, a single column within the class' database table will be created. The name of the column corresponds to the name of the instance variable. However, the standard preset dissects the original variable name into the separate sub words and connects them with an underscore. A variable named 'userName', for example, is thereby converted to the column name 'user_name'. This is required to provide simple compatibility with most other O/R mappers for dynamic programming language environments, such as ActiveRecord for Ruby on Rails (see chapter 5).

The mapping of the data types is implemented within class side methods that are named `sqsType`. For all classes that are trivially mappable, this method has been implemented and

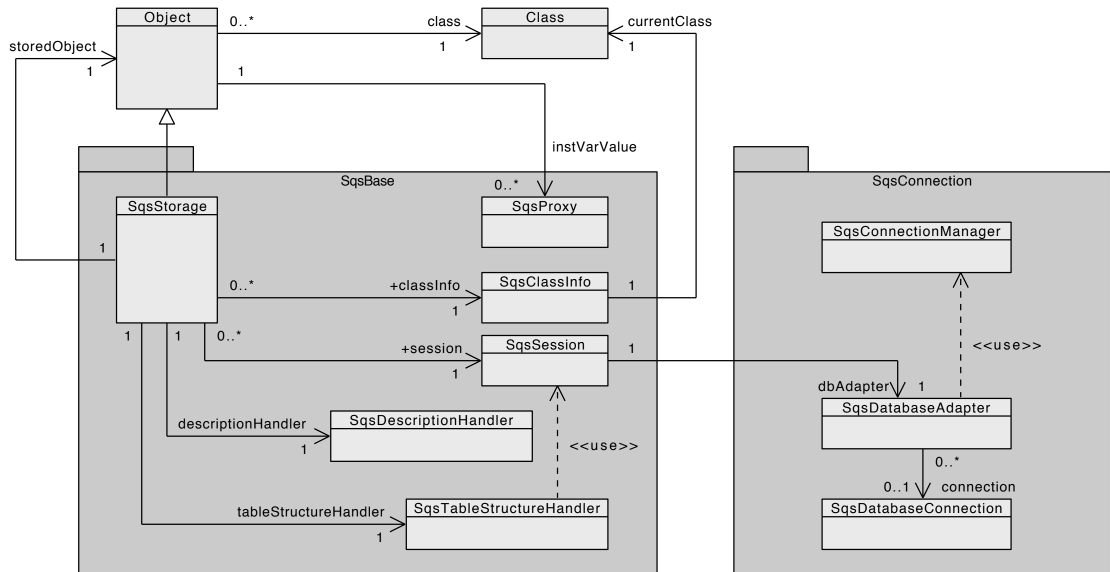


Figure 4.1: Overview of SqueakSave System Classes.

returns a SqueakSave internal string representation of the according SQL type. If the columns have to be created, those internal representations are translated by the `SqsDatabaseAdapter` classes into the specific values that are required by the current database servers' SQL implementation. Types with variable lengths, like strings, are additionally enriched with the information about the current length of the respective object. Hence, a string of length 50 will not only be mapped to TEXT or VARCHAR, but VARCHAR(50).

Non-trivial attribute types are mapped by a foreign key reference to the corresponding entry in the table that represents the class of the respective object. The reference will always point to the table of the base class, i.e., the first class in the inheritance chain below `Object` or a class that is marked like depicted in section 3.3.3, which is especially important in class table inheritance structures. They are created in such a way, that a separate table for each subclass is created and only contains the attributes that are defined within this class. Therefore, a foreign key constraint pointing to only such a sub table would prevent the possibility to reference objects of super or subclasses.

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 may-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 supposed to be following 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 applications object memory.

The second problem is the inversion of the logical association direction from the object model

to the relational structure [38]. Instead of the collection owner pointing to the values of the collection, the elements within that collection would reference their owner. This fact is also problematic with regards to the usage of objects within many collections in different classes of objects. It would be required to add a new table column for every reference to those objects.

The created join tables contain a field that references 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 collections with a strict internal order. 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 the collection elements will be replaced with a field of the respective value, that directly stores them within the join table.

All of the aforementioned mapping rules are manifested within the mapping descriptions. They include the derived column names and types, as well as the descriptions for more complex structures like join tables.

Description Updates In order to reflect changes within the object model in the database structure, SqueakSave uses Squeak's meta-programming and reflective capabilities to update the previously created descriptions. The Petri net depicted in figure 4.2 presents the workflow of the update procedure. Shortly summarized, each instance variable value is checked against previously created descriptions and they are persisted in terms of a newly created mapping description only if changes to the relational structure would be necessary.

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 (see chapter 3), this behavior could be observed if an `Admin` object is the current value of an attribute that was previously pointing to general `User` objects.

With regards to collections the behavior slightly differs. Not only the type of collection is of importance here, but also the base type of objects within that collection. Thus, a description alteration will be performed if, for example, an `OrderedCollection` has been transformed into a `Set` or if it contains values of a different base class than its predecessor.

The SQL type inspection is necessary to detect values of the same class, that require a different encoding within the database. While, for example, strings with a length shorter than 255 characters can be stored in columns of the type VARCHAR, longer ones require the database field to be of type TEXT. This is an important feature with regards to the optimization of the created database schema. Since accessing TEXT elements is vastly slower than querying VARCHAR fields [51], SqueakSave strives to only use this fields if the application indeed requires them to store its data.

The final check for the addition or removal of foreign key constraints is especially important for associated complex attributes and owned collections. If, for example, the class of a direct associated object changes, or collections include different values, the foreign keys have to be altered, as well. They need to point to different tables or have to be removed completely, if a

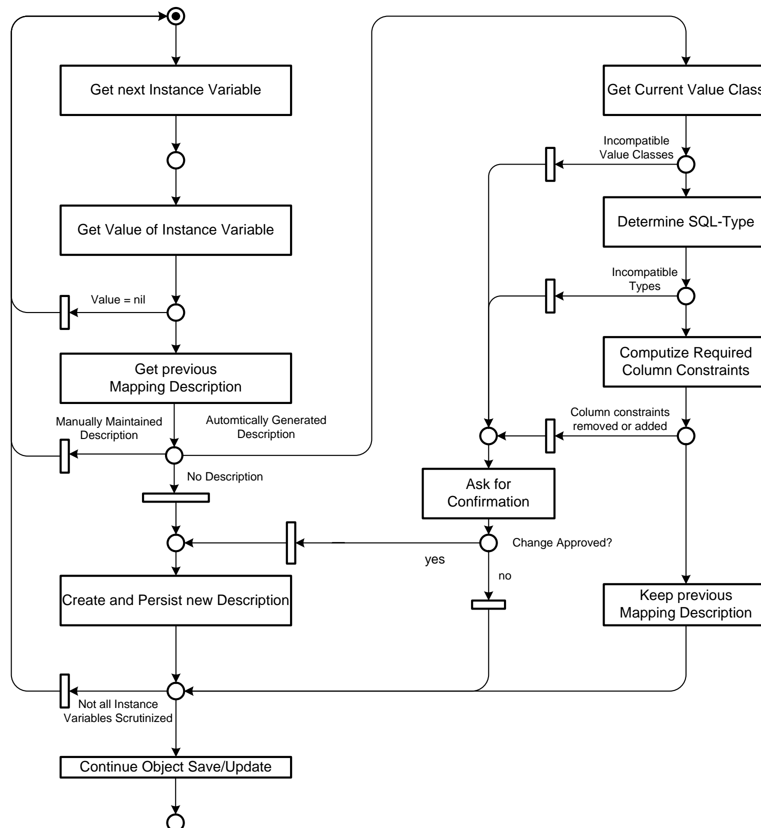


Figure 4.2: Workflow of Automatic OR-mapping description updates.

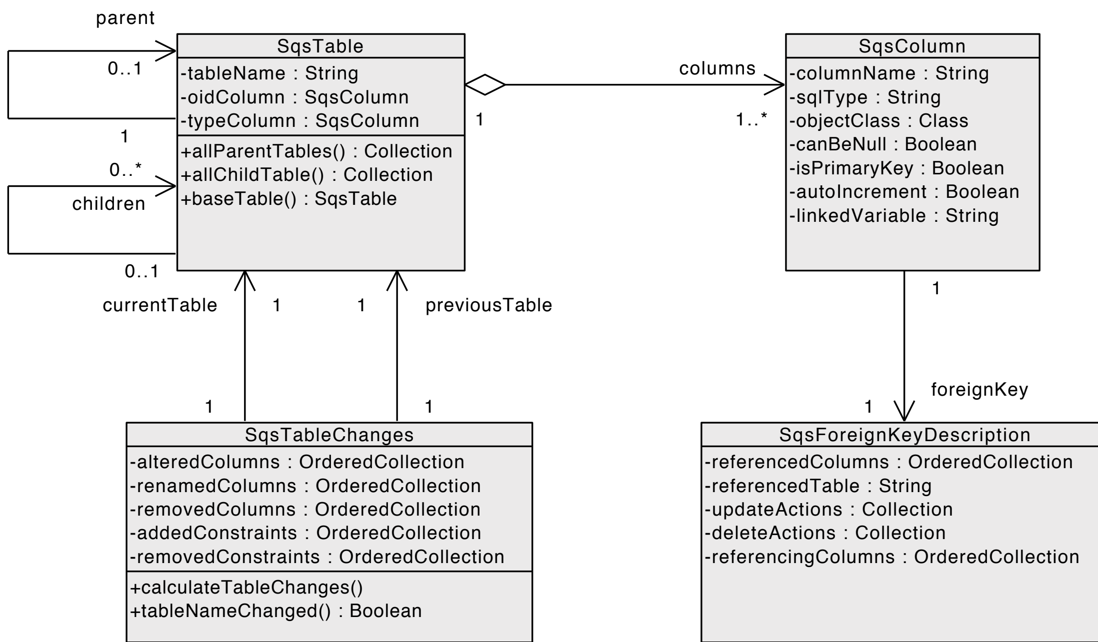


Figure 4.3: Table Description Classes.

complex association is now referencing simple objects.

4.1.2 Table Structure Adaption

As previously described, SqueakSave incorporates means to support different standard patterns to map object structures to relational schemas. This behavior is achieved by an abstraction from the actual table structure, that can be traversed in a generic manner. As depicted in figure 4.3 each table can have a number of columns, foreign key constraints and and child tables. In addition, each child table also includes a references to its parent table.

Utilizing this behavior, it is possible to represent table structures, whereas the actual values of object attributes are distributed within an arbitrary number of sub tables. In conjunction with the foreign key constraints, that manifest those relations on a database level and ensure referential integrity between the corresponding table rows, it is possible to create normalized database schemas. However, in order to achieve this behavior, custom table structure handlers have to be implemented (ref. chapter 4.4.2).

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. It therefore provides the foundation for the table structure updates that are carried out during the execution of the `save` method. The previously present table models are stored in a cache within the `SqsTableStructureHandler` class and compared against the versions represented by the currently available mapping descriptions.

If tables differ within their structure, the updates are carried out in an order that guarantees

Object

+save()
+save(session : SqsSession)
+flatSave()
+flatSave(session : SqsSession)
+deepSave()
+deepSave(session : SqsSession)
+saveToLevel(level : Integer)
+saveToLevel(level : Integer, session : SqsSession)
+rollback()
+rollback(session : SqsSession)
+destroy()
+oid() : Integer

Figure 4.4: Additions to the Object Protocol.

the avoidance of query errors due to inconsistent table alterations:

1. Removal of superfluous constraints,
2. Renaming of the according class table,
3. Renaming of columns, whose names have been altered by framework users,
4. Alteration of column types,
5. Addition of new columns,
6. Removal of unnecessary columns,
7. Addition of newly created constraints.

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

4.1.3 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. Due to the requirement of not altering existing class definitions, the SqueakSave framework relies on the `SqsStorage` class as a decorator [21] that handles persistence-related operations. As mentioned in section 3.2.2, the protocol of the `Object` class has been extended with methods that support basic CRUD functionality. Figure 4.4 presents those custom additions.

All of the aforementioned calls will be internally delegated to an instance of `SqsStorage`. For each object that is present within the image, a unique `SqsStorage` instance is created. Due to a caching mechanism that is utilizing weak references [25], the respective instances are only available as long as the base object is not subject to garbage collection.

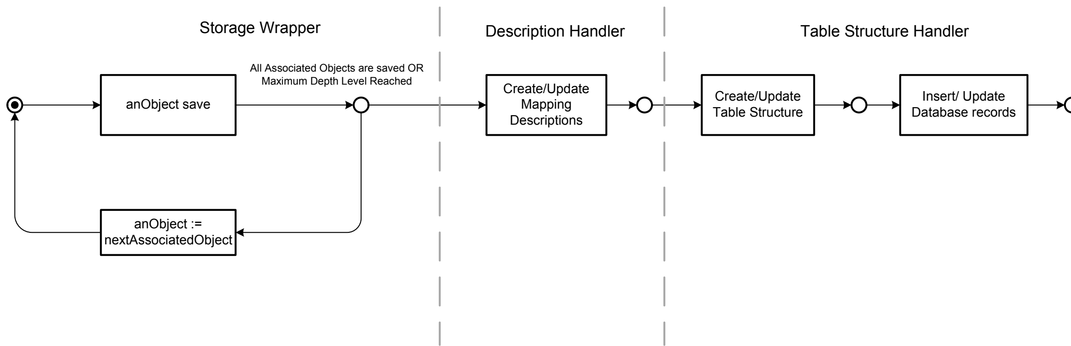


Figure 4.5: Internal Workflow of the Save-Operation.

In addition to the decorator, the framework will also create a unique object id for each persisted object. Those unique identifiers are required to couple an object with its database representation and, accordingly, enable references between objects on the database level [3].

By tightly coupling the decorator instances to the decorated objects, it is also possible to handle recursive calls of the `save` method. As presented in figure 4.5, the decorator instances will only try to store associated objects, if the object itself has not already been processed within the current operation. An internal flag is set at the first traversal, and if cyclic references lead to an object again, only changes to instance variables and owned collections will be examined.

The example of two cyclicly coupled objects, that are stored within the same `save` call for the first time, will clarify this behavior: The algorithm first writes all attributes of the main object into the database. However, since the referenced object has not been subject to persistence operations before, it does not have a unique object id assigned at this point. Hence, the corresponding entry within the database will be a NULL value. Afterwards, the referenced object will be saved, and because the root object has a valid object id, the database record will, accordingly, point to this value. As the framework now tries to store the root object again, because it is referenced by the second object, it will not call the `save` method again on all referenced objects, but only update the corresponding database entry of the root object with the now present object id of the second object.

Object Proxies For performance and framework internal reasons, instances of `SqsProxy` are inserted into the decorated objects. Proxy objects are distinguished between proxies for directly associated objects and those representing collections.

The proxies for directly associated objects, like the account data of a user in the example application (see chapter 3), are necessary to avoid an eager loading of the entire object graph upon the creation of query results. If, for example, a user object is returned for a search, the corresponding account data would have to be loaded, as well. In order to improve the performance of queries, the proxy objects are inserted instead of the concrete associated objects. The proxies are initialized with all required information to trigger the loading of the depicted object if the applications accesses them. All calls to the proxy objects, except for those that are defined on `ProtoObject`, are delegated to the loaded instances. Thereby the insertion of the proxies re-

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

4.2 Utility Classes

To realize the previously described behavior of the framework, the core classes have to utilize a number of utility classes that give them the ability to cache objects, obtain database connection, or adapt their behavior to custom configurations. The following sections describe the most important utility classes of SqueakSave.

4.2.1 Object Caches

Object caches are not only necessary to enhance the performance of the framework, but also an integral asset for the realization of adding persistence merely transparently to objects. Since the object structure shall not be altered, their internally assigned unique object identifiers cannot be stored within the objects itself. As depicted in section 4.1.3, a unique decorator is assigned to each object during its lifetime. Therefore the obvious choice for implementing the object id storage is the introduction of an instance variable to the `SqsStorage` objects.

The straightforward solution to this caching issue would be the creation of a dictionary, whereas the objects are used as indexes that point to their respective decorators. Unfortunately, the inefficient handling of large collections within Squeak would lead to a decreased degree of scalability of the framework for an increasing amount of stored objects. Therefore, the implementation builds up the cache in a staggered manner. Firstly, the class of the respective objects is used for a preselection of the corresponding sub caches. Secondly, within the sub caches the decorators are stored within a B-Tree structure [7] that utilizes their Squeak-internal object hashes to calculate the exact position of the decorator reference within the cache.

In addition to the usage of caches to store object ids without object model or inheritance structure alteration, performance optimization also requires this feature. To avoid the rebuilding of objects that have been already the result of a query, or have been instantiated within the application 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.

4.2.2 Database Connection Handling

The database adapters encapsulate the SQL query generation according to the specifications of the respective RDBMS. In order to execute those queries, the adapters rely on instances of `SqsDatabaseConnection`. SqueakSave database connection objects conceal differences between the connection objects supplied by the different database access drivers (see section 4.4.3).

The physical connection to the database 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 been an option that had highly 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, especially, smaller queries. Therefore, SqueakSave implements a centralized connection pool. This pool is maintained by the singleton instance of the `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 is guarding the insertion and retrieval processes with a semaphore. 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. The detailed workflow of the connection retrieval and return is depicted in figure 4.6

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.

4.2.3 Configuration and Customization

Customization of the framework is possible by the means presented in chapter 3.3.1. In order to integrate the custom settings into the workflows that are performed by the framework, it is necessary to implement each aspect with close attention to session utilization, as they include the currently active configuration. Session objects are available to each framework actor by two means: Indirect access through a reference to the decorator of the currently investigated object and an explicit request to the `SqsSessionManager`, that will return the session used within the current thread of control. The former method is available for classes such as `SqsDescriptionHandler` and `SqsTableStructureHandler`, while the latter is reserved for queries, since they are not coupled to a specific decorator instance.

With the session, and accordingly the configuration, at hand, the actors can adopt certain behavior to the values specified by the developers. This most obviously manifests itself within the aforementioned description and table structure handlers, as they are created in a factory pattern [21] fashion based on the chosen table inheritance model and the respective description handler class. Additionally, the configuration influences their workflow for table creation and

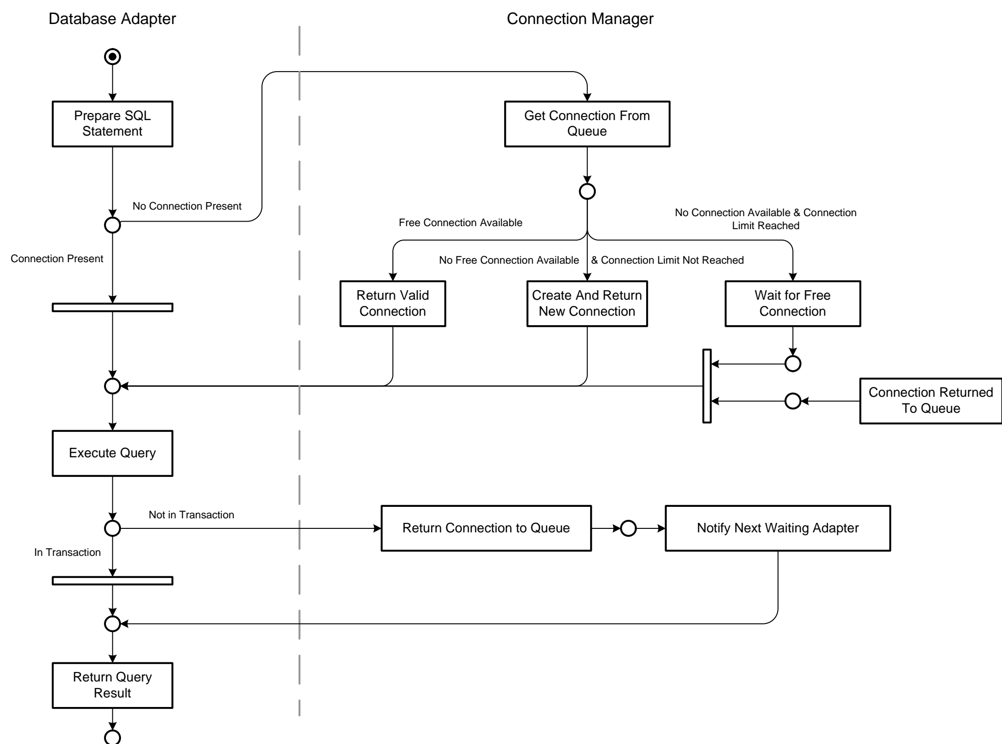


Figure 4.6: Retrieval of Database Connections.

description updates by stipulating values for certain table fields or rules about developer approval for description alteration.

4.3 Query Generation

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

4.3.1 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[1]. 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 4.7. Each of those classes imitates the protocol of basic system classes such as `Integer` or `String`. But instead of of delivering the result for each operation, the methods gradually fill the `whereBuffer` attribute with the SQL equivalents of the respective operations. Listing 4.2 presents the SQL WHERE statement that is generated for a sample query (listing 4.1).

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

Listing 4.1: Language-Native Query Before Translation to SQL.

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

Listing 4.2: SQL WHERE Statement Generated from Language-Native Query.

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

[1] http://www.squeaksource.com/REServe.html

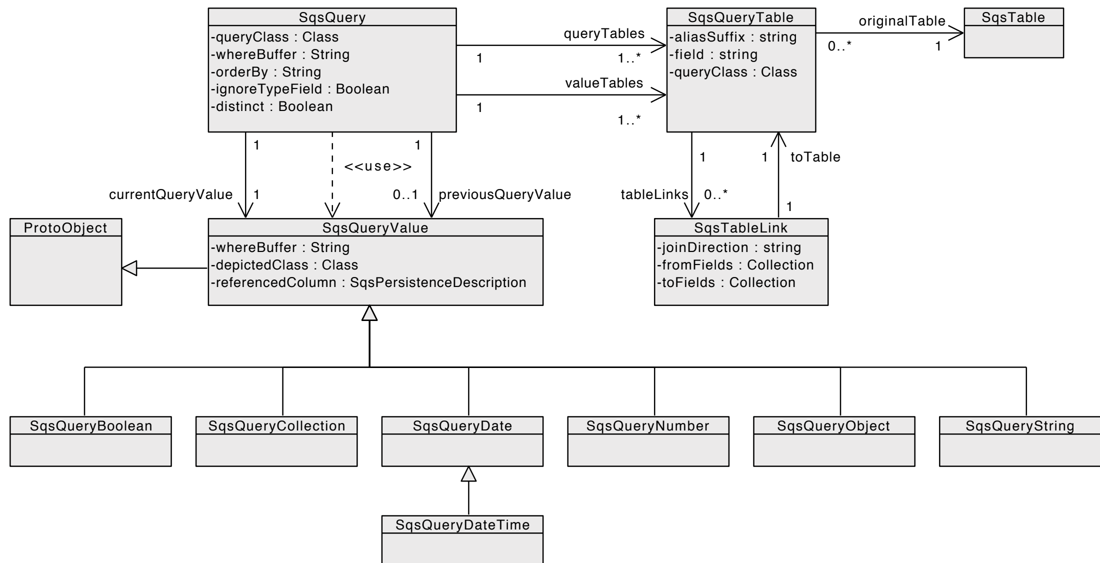


Figure 4.7: Collection Protocol Emulation Classes.

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 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.

4.3.2 Convention Based Query Methods

The dynamic convention-based finder methods, presented in chapter 3.2.3, could be implemented separated from the collection protocol emulation. However, considering the requirement for extensibility of the framework, their implementation is based on the collection protocol emulation, as well. 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 4.4 depicts the block-closures created from a dynamic finder method (listing 4.3).

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

Listing 4.3: Dynamic-Finder Method Before Conversion into Block-Closure.

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

Listing 4.4: Block-Closure Generated from Dynamic-Finder Method.

4.4 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 object-relational mapping flavors. Therefore, the classes that are responsible for the realization of the corresponding behavior have been implemented in ways that ought to simplify the development of custom framework extensions.

4.4.1 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.

As presented in figure 4.8, only two methods have to be implemented in order to create new mapping description handlers. The `sqsDescriptionFor:instVarName` method 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.

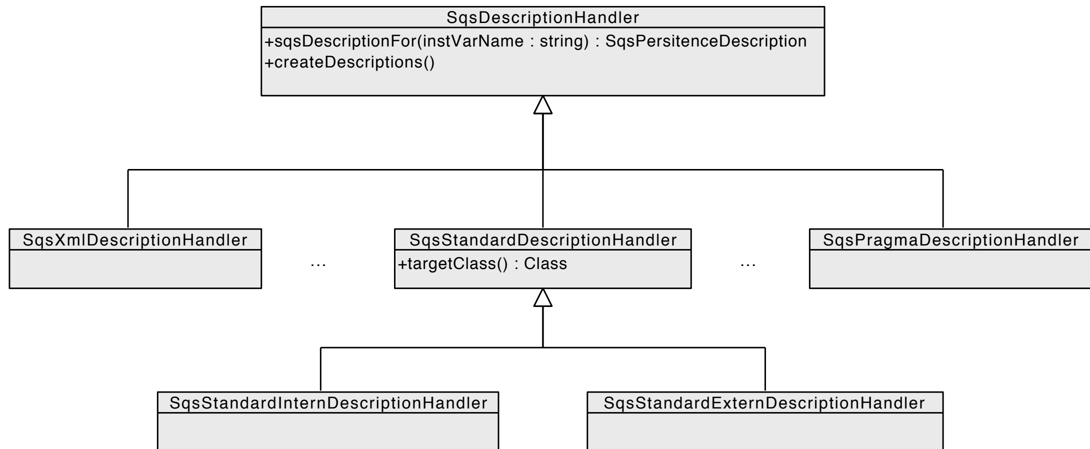


Figure 4.8: Description Handler Inheritance.

4.4.2 Table Structures

In order to provide the means to create new table structures that do not follow the standard patterns of class, single or concrete table inheritance, it is necessary to implement a subclass of `SqsTableStructureHandler`. The only method that has to be implemented regardless of the chosen mapping model is `classTable`. It has to deliver an `SqsTable` object that includes all columns belonging to that table and additionally all its sub tables and super tables.

The implementation of this method, however, is free to re-order table columns according to any criteria of choice. Therefore, it could be possible to create a table structure handler that distinctively utilizes in-depth knowledge about the internal storage mechanisms of a given RDBMS. This could lead to optimized, but less comprehensible database schemas, that might not directly represent the object model, but adhere to relational database normalization rules.

4.4.3 Database Adapters

The most obvious extension point for any given 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 a 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[2] or OpenDBX[3]. 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 localization of methods within the protocol of `SqsDatabaseAdapter`, that have to be overridden in order to provide a working adapter implementation for a certain RDBMS is rather difficult. 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, and thereby tries to minimize number of methods that have to be overwritten.

4.5 Summary

The main requirements for the implementation have been the realization of automatic updates, language-native queries, and extensibility of the framework. Throughout this chapter necessary design decisions for the implementation of this behavior have been presented. Automatic updates are implemented by a copious algorithm that supposedly covers all possible changes to the object models and therefore dependably and only in unavoidable scenarios updates the existing mapping descriptions.

Language-native queries have been implemented by a block-closure analysis system, that can handle deep object graph structures and numerous standard operations on simple data types, as well as all accessor methods on complex objects. While it could be extended with means to fully analyze all available method calls in order to provide fully transparent persistence [54], the current implementation suffices for the most common usage patterns.

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

[2] `http://support.microsoft.com/kb/110093`
[3] `http://www.linuxnetworks.de/opendbx`

5 Evaluation

The main focus of the implementation of SqueakSave was the support of fast-evolving object models and 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 [4], and thus the implemented framework has to be evaluated with regards to both aspects. The following chapter provides benchmark results for SqueakSave and three other O/R mapping frameworks. Additionally, the production and development modes are compared and conclusions are drawn regarding performance bottlenecks and possible optimizations. The chapter concludes with an example-based evaluation of the interoperability between SqueakSave and ActiveRecord for Ruby on Rails.

5.1 Performance

Numerous benchmarks exists to measure the performance of object persistence technologies. The BUCKY [12] or the BORD benchmark [35], for example, are especially designed to analyze the performance of object-relational systems. Other approaches, like the OO7 Benchmark [11], 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 and merely add it as an aspect to the application that requires no additional adaption of the existing object model. Therefore, the OO7 Benchmark is utilized for performance measurements.

The implementation used for this comparison is based on the Java version[1] of the original benchmark, which was written in C. Porting this benchmark to Java has been carried out by Zyl et al. in order to compare the performance of object-relational mappers and object-oriented databases [56]. Since neither the original benchmark presentation, nor the aforementioned comparison depicts the class structure used within the benchmark, figure 5.1 presents it to visualize the complexity of the model.

All benchmarks 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. To minimize the impact of temporary changes within the execution environment, each benchmark was run 100 times. The measurement results represent the median of all retrieved timings.

[1] http://sourceforge.net/projects/oo7/

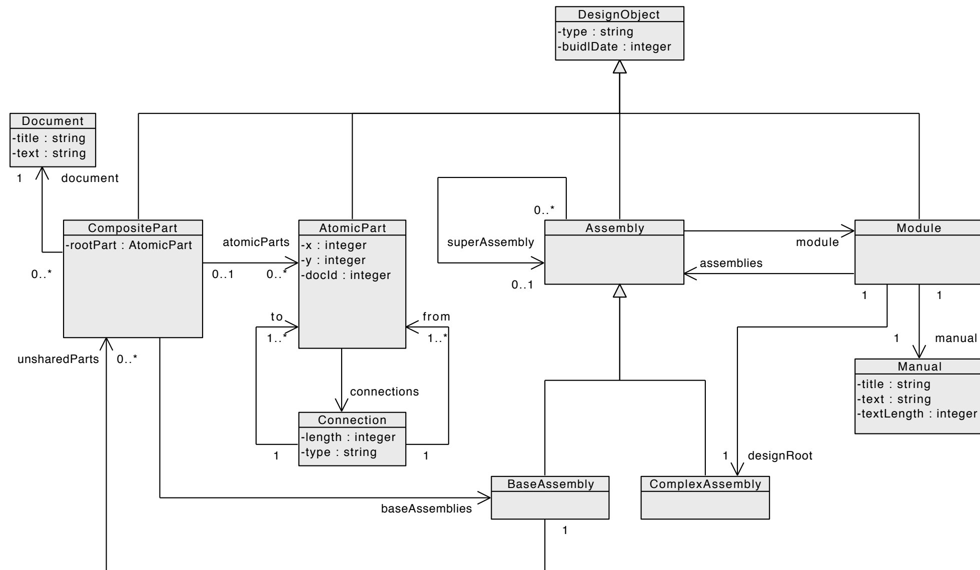


Figure 5.1: OO7 Benchmark Class Model.

5.1.1 Comparison with other Object-Relational Mappers

To evaluate the overall query and object graph traversal performance of SqueakSave, the benchmark has been additionally performed with two other object-relational mapping frameworks for dynamic programming environments and an implementation for a statically-typed programming language.

Since platform specific limitations and performance bottlenecks, such as overall inferior execution speed or subpar implementations of viable system classes, impede objective measurements, the comparison has to include a comparable system implemented within Squeak: The generic lightweight object-relational persistence framework (GLORP) [33].

The other O/R mappers used for this comparison are ActiveRecord for Ruby on Rails and Hibernate [17] for Java. As previously mentioned, the benchmark results for those frameworks cannot be objectively compared to the results for SqueakSave and GLORP because of the major differences between the respective execution environments. Elaborated implementation details such as the Just-In-Time compiler of the Java HotSpot VM or natively compiled libraries for RDBMS access provide considerable advantages which, of course, have an impact on the measurement results.

While those comparisons provide no direct indicator for the performance of SqueakSave, they will allow to deduce fields of application where the combination of SqueakSave and Squeak might be advantageous for developers. Additionally, the implementation paradigms of ActiveRecord and Hibernate correlate with the ones underlying SqueakSave and GLORP. ActiveRecord and SqueakSave require explicit save operations to store or update objects, while Hibernate and GLORP are 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 re-

gards to performance.

Following versions of the frameworks and their corresponding PostgreSQL client implementations have been used:

- SqueakSave: Revision 107, PostgreSql Client 1.0 (Smalltalk implementation), Squeak 3.10 image, VM version 3.8.18.
- Hibernate: Version 3.2, JDBC3 for PostgreSql 8.3 (Java implementation), Java HotSpot VM build 1.5.0_16-133.
- ActiveRecord: Version 2.2.2, PostgreSql Gem 0.7.9 (natively compiled library), Ruby VM version 1.8.7 (patch level 72).
- GLORP: Version 0.4.169 , PostgreSql Client 1.0 (Smalltalk implementation), Squeak 3.10 image, VM version 3.8.18.

To further avoid influences on the measured timings, each system was set-up to its 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 magnitudes. Each of them highly increases the amount of stored objects and also the connections between them. The third-largest version of the benchmark was used, since it reflects the intended application area for the SquakSave framework in terms of database usage. It includes approximately 10.000 atomic parts and 30.000 connections between them and thus reflects the database payload of small to mid-sized applications.

Figure 5.2 presents the overall creation time for the database schema that is required to perform the OO7 Benchmark. It is evident that Hibernate and GLORP outperform the other frameworks. 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 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 (ref. chapter 3) and not execution speed.

Query Performance Comparison The queries performed during the OO7 benchmark continuously increase in terms of complexity and result count:

- Query 1 sequentially retrieves 10 arbitrarily chosen atomic parts, thus each single query only returns one result and operates on the indexed *id* field.

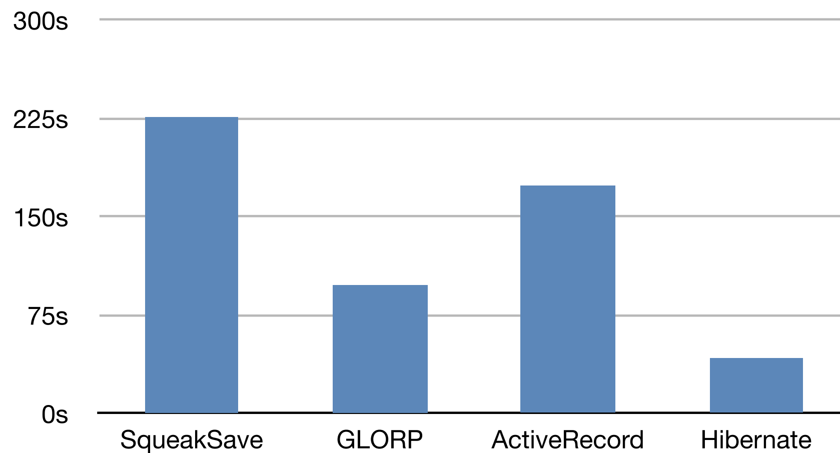
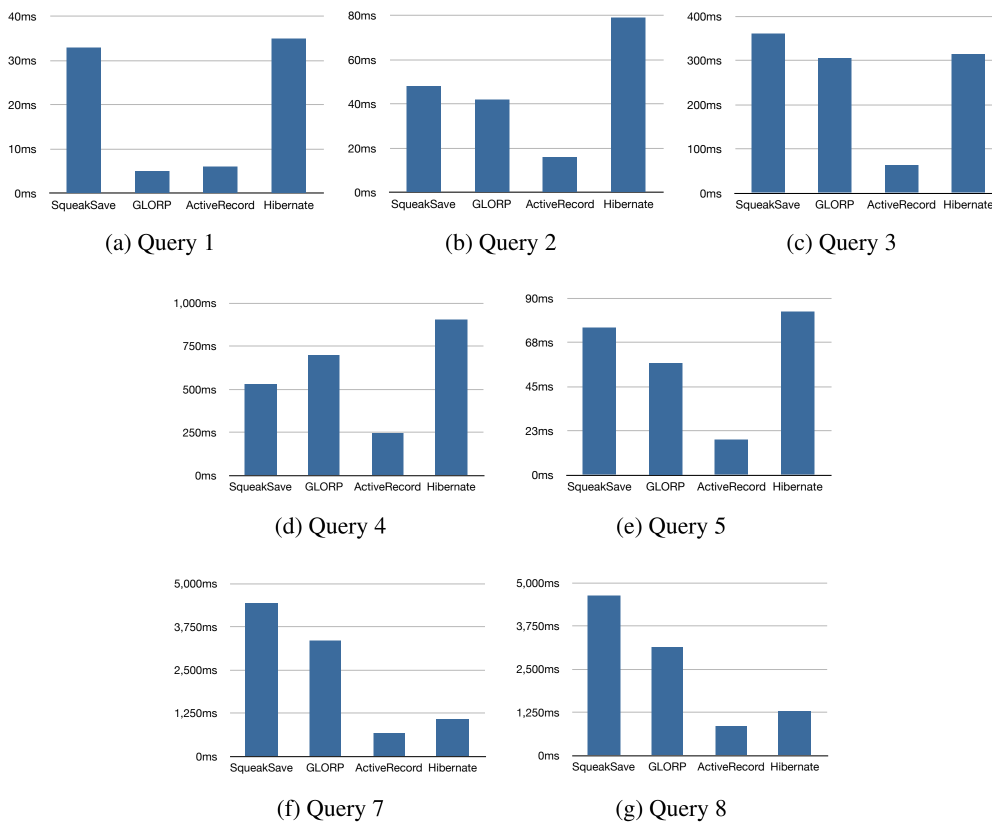


Figure 5.2: OO7 Benchmark - Database Creation Times for Different O/R-Mappers.

- Query 2 calculates a very strict date threshold which is used to find all atomic parts whose build date exceeds the selected boundary. It returns approximately 100 results.
- Query 3 is identical to query two, however, the date threshold is slightly lower and leads to about 1000 atomic parts fulfilling the search criteria.
- Query 4 selects 100 random documents from the database and consecutively finds base assemblies, which have at least one unshared part that points to the particular document. The query is the first one to require an explicit join between two database tables. For each of the 100 sub queries the search delivers an average of two results
- Query 5 finds all base assemblies that reference unshared parts with a build date that exceeds their own. It delivers approximately 250 results, depending on the randomly assigned build dates
- Query 6 has been omitted by the developers of the Java OO7 version because it was not assumed to deliver any meaningful results. Hence, it has been omitted within this comparison, too.
- Query 7 creates the highest result count (10000), as it queries for all atomic parts without any further restrictions.
- Query 8 detects all pairs of documents and atomic parts where the randomly assigned *docId* property of atomic parts actually points to a valid document database entry

The original implementation of the benchmark had to be slightly altered in order to provide meaningful results for the eighth query. Since the *docId* attribute of atomic parts is not a direct association to an actual document, but filled with random numbers between one and ten during the database creation, it had to be ensured that matching document objects are found during this query. In the original version, however, the internal object id assignment of Hibernate has been



(a) Query 1 (b) Query 2 (c) Query 3

(d) Query 4 (e) Query 5

(f) Query 7 (g) Query 8

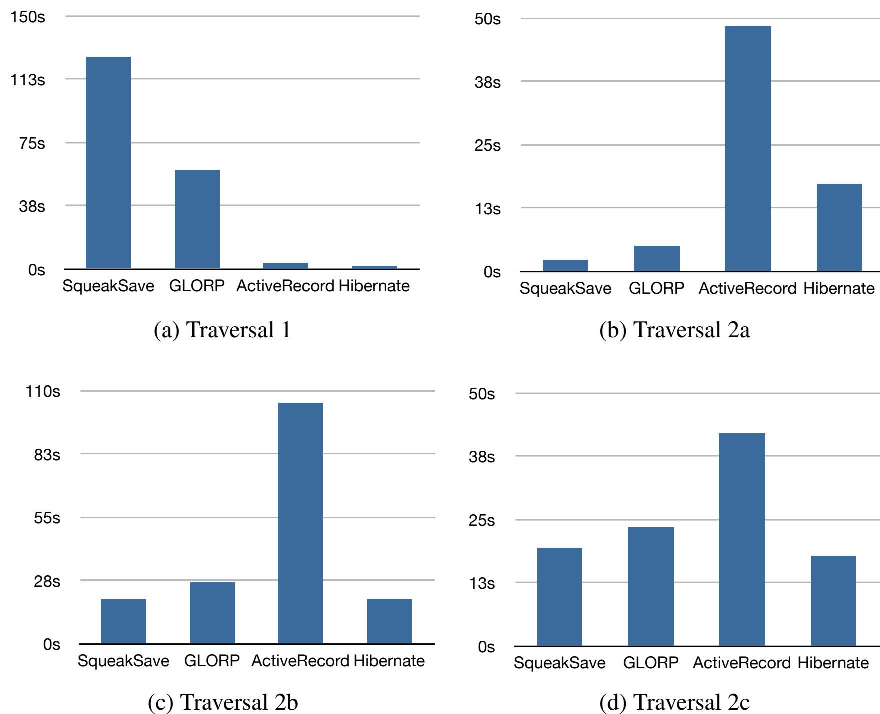
Figure 5.3: OO7 Benchmark - Query Times.

used, which not utilizes the database specific ids, but sequentially assigned integer values for all benchmark objects. This led to only one document within and id range of one to ten, and, accordingly, lowered the result-count for the search.

The query times presented in figure 5.3 show that with regards to query performance, the combination of ActiveRecord and Ruby is generally faster than the other tested framework-environment pairings. Only in query one, GLORP was able to have a slight edge over ActiveRecord. However, this is not a result of superior query performance, but a consequence of optimistic caching. Instead of performing the query on the database, the results are delivered directly from the cache. While this obviously increases the query performance, it is also prone to errors. If the respective object had been removed from the database within another session, the query would return an object that no longer exists in persisted space.

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

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



(a) Traversal 1

(b) Traversal 2a

(c) Traversal 2b

(d) Traversal 2c

Figure 5.4: OO7 Benchmark - Traversal Times.

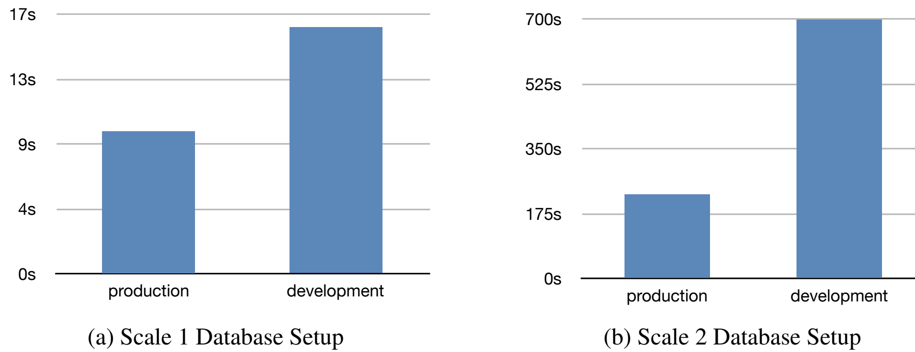
larger. Section 5.1.3 will provide some reasons for this behavior.

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

Traversal Performance Comparison 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.

To obtain objective measurements, 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.



(a) Scale 1 Database Setup
(b) Scale 2 Database Setup

Figure 5.5: OO7 Benchmark - Database Creation Times for SqueakSave Modes.

The results depicted in figure 5.4 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.

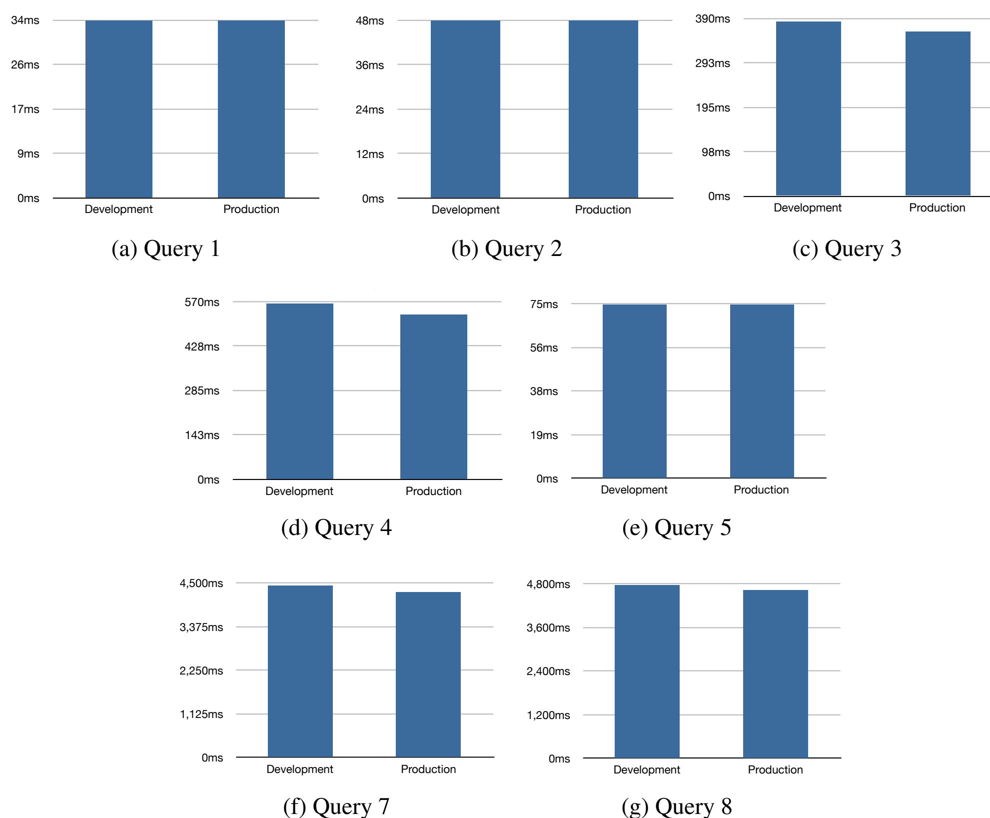
The traversal times in the following tests obviously increase since the atomic parts are not only being traversed, but also updated. Therefore, it was expected that the advantage of SqueakSave slightly diminishes. However, the traversal times in those tests still show, that for the traversal of previously 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 Hibernate and 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.

5.1.2 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 5.5 depicts the creation times for the small and tiny database layout. It can be clearly



(a) Query 1 (b) Query 2 (c) Query 3

(d) Query 4 (e) Query 5

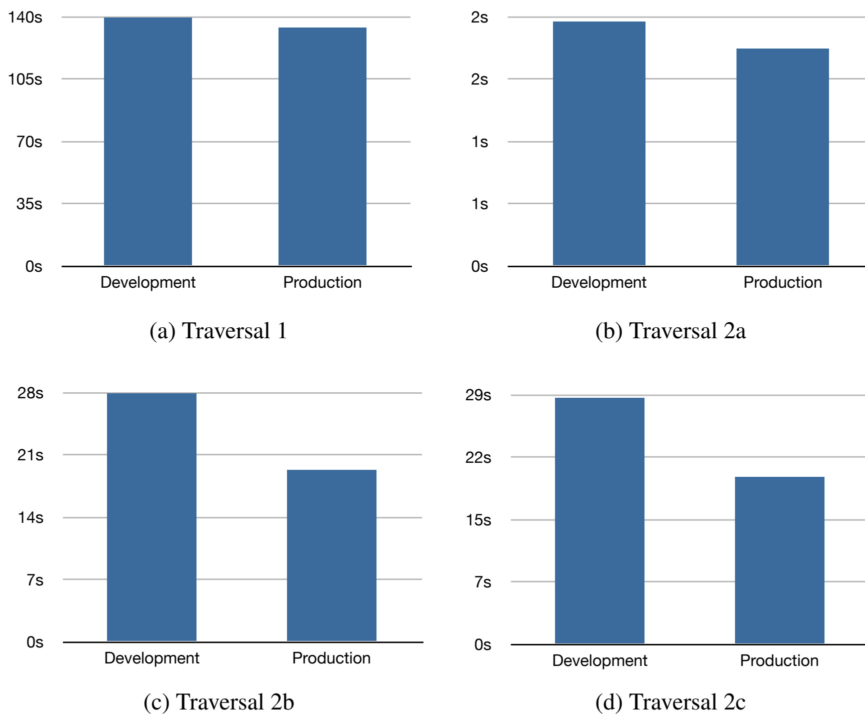
(f) Query 7 (g) Query 8

Figure 5.6: OO7 Benchmark - Query Times for SqueakSave Modes.

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 in a linear manner.

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

The query performance does not differ between development and production modes, since the synchronization between object model and database representation only takes places during object saving and, accordingly, does not affect the search queries (ref. figure 5.6).



(a) Traversal 1

(b) Traversal 2a

(c) Traversal 2b

(d) Traversal 2c

Figure 5.7: OO7 Benchmark - Traversal Times for SqueakSave Modes.

During the traversal measurements, however, the previously observed differences still apply (ref. figure 5.7). While the first traversal is barely affected by the current execution mode, changes to the object model (ref. 5.7c, 5.7d) are performed much faster within 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.

5.1.3 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 during the automatic retrieval of configuration objects from the respective configuration classes. This is a direct consequence of the fact that Squeak does not incorporate categories as first class objects, and thus a time-consuming lookup for the respective classes has to be performed

- The storing of object ids within a distinctive caches does not vastly affect the execution speed. However, upon large scale operations, such as the creation of the benchmark

database, the impact becomes no longer diminishable, 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 ineffective 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.

- The fine-grained `save` operations provide a viable means to control the execution of database inserts 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, which allow for calling the save method on the root object of an object graph and combine the insert and update operations to as few SQL queries, as possible

- Eager loading of objects is an important aspect with regards to the traversal of object graphs. Therefore, 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

5.2 Interoperability

Adding persistence functionality to existing applications by integrating SqueakSave has been shown to be rather effortless in chapter 3. In order to also provide a brief evaluation of the interoperability capabilities of the framework, the sample application has been implemented as well in SqueakSave, as in ActiveRecord.

The first use case for interoperability is the application database creation by SqueakSave and a subsequent usage by ActiveRecord applications. The general schema adheres to ActiveRecord naming conventions, and, accordingly, creating the respective model classes is sufficient in terms of simple value representation. In order to add one-to-one associations, the `belongs_to` macro has to be added to the model, that holds the reference to another class. Within the sample application this would, for example, be necessary within the `User` class.

In order to depict the created table structure within its object model, one-to-many relations impose some problems to current versions of ActiveRecord. The `has_and_belongs_to_many` macro would have adequately modeled the fact, that all to-many relations are materialized by means of join tables. Unfortunately, it has been marked deprecated [18]. Therefore it is necessary to create an explicit join model for each of the one-to-many relations within the object model (e.g. `Blog-BlogPost` or `BlogPost-Comment`).

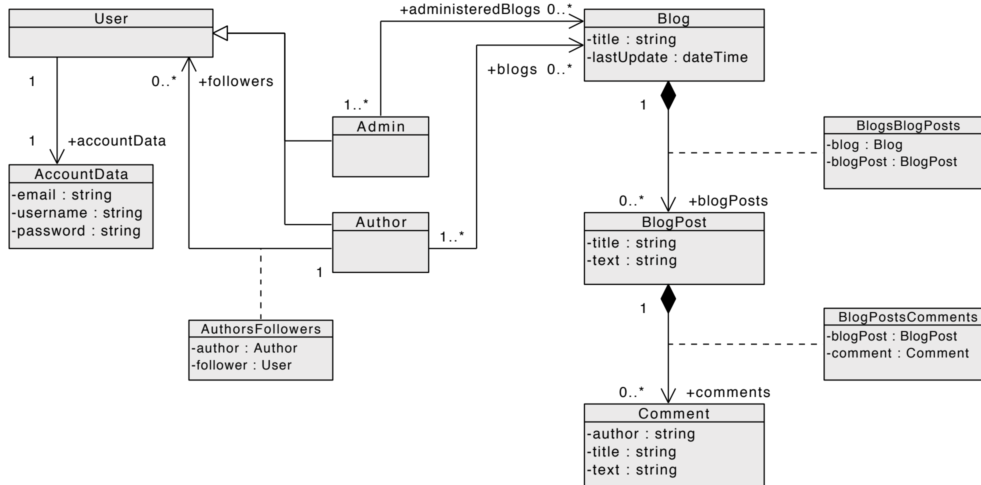


Figure 5.8: Class Structure of the Example Application with ActiveRecord.

This leads to a slight discrepancy between the class structure of the SqueakSave application (figure 3.1) and the required classes for the Ruby solution (figure 5.8).

If the database model has been created by ActiveRecord developers and with explicitly defined one-to-many relations, the mappings within the SqueakSave application have to be slightly altered. It is necessary to change the `SqsCollectionDescription` for the corresponding attributes in a way that the name of the referenced table is not the one of the join table, but of the table of the respective referenced objects. The descriptions for other attributes can be inferred, since SqueakSave by default utilizes the same naming conventions as ActiveRecord.

So, basically, in order to overcome the mismatch between SqueakSave and ActiveRecord, it would only be necessary to model all to-many associations by using join models, instead of maintaining back references that, in addition to not being faster than join tables, also inverse the object model associations within the database.

5.3 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.

6 Related Work

In this chapter related work is presented distinguished by the environment, solutions for object persistence have to deal with. As for object-relational mappers, this requires a distinction between dynamic and static object relational mappers. While the former require means to handle systems that do not support compile-time type checking and thus need to adopt to the possibility of dynamically evolving data schemas, the latter might utilize static source code analysis in order to create mappings between object models and their relational representation.

Additionally, object databases will be presented, that not only adhere to the principles outlined in section 2.2.3. but additionally strive to provide a degree of interoperability with object-relational solutions.

6.1 Static Object-Relational Mappers

Lodhi et. al [38] propose a simple means to keep object models consistent with database schemas by utilizing a Rational Rose plug-in, in order to compile standard mappings from previously modeled application classes. While this solution automates the task of O/R mapping creation, the type system remains static, even when compiled to dynamically-typed object-oriented languages. Additionally, the up-front modeling approach does not cover classes created or altered during application run-time.

A, to some degree, similar approach has been presented by Melnik et. al [40]. It uses declarative mappings between objects and relational database structures and compiles them into bi-directional views, that not only provide means to operate on persisted data but additionally implement consistency between database entries and application objects upon direct changes to the database. The presented system is implemented within the ADO.net framework [1].

Providing persistence as an aspect, and, thus, transparently and interchangeably to applications, has been proposed by Rashid et. al [47]. While they conclude that it is possible do design systems in such a way, it is also stated, that persistence to some degree always influences architectural decisions, especially regarding data-consumer components and the query support of the underlying database model.

Hibernate[1] is the most popular open source O/R mapping framework for Java [17]. It offers comprehensive and exchangeable mapping descriptions in different formats and a unique query language (HQL). Additionally, it is possible to automatically create mappings for existing applications by using an extensive tool set.

Hibernate also provides the foundation for other O/R mapping solutions such as NHibernate[2],

[1] http://www.hibernate.org
[2] http://www.nhibernate.org

a direct port to the .Net framework, and Castle ActiveRecord[3]. The latter is an adoption of the Active Record pattern [20] and accordingly provides an additional layer on top of Hibernate for a more convenient access.

6.2 Dynamic Object-Relational Mappers

Not only does Hibernate provide the foundation for O/R mappers within statically-typed languages, but also for the Grails Object-Relational Mapper (GORM)[4], which is executed in a dynamically-typed object oriented environment - Groovy[30]. GORM provides a minimalist API do describe the mappings between objects and relational constructs on top of the extensive description capabilities of Hibernate. In addition to the possibility of using HQL for queries, GORM introduces a language-native query API that utilizes dynamic methods to generate the SQL statements for each method call.

ActiveRecord for Ruby on Rails [18] is a database schema-driven O/R mapping solution that adheres to the convention over configuration (CoC) principle [49] . 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 within 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[5], another object-relation mapping framework written in Ruby, 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 [33] provides object-relational persistence by heavy utilization of meta descriptions. The descriptions have to 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 it 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[6] 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[7] project, that removed the automatic table creation, but

[3] `http://www.castleproject.org/ActiveRecord/`
[4] `http://www.grails.org/GORM`
[5] `http://www.datamapper.org`
[6] `http://www.squeaksource.com/IOSPersistent.html`
[7] `http://www.squeaksource.com/ReServe.html`

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.

6.3 Object Databases

The Gemstone-project [10] provides persistence in an almost transparent manner to applications. However, it requires an extensive environment in order to be applied as a persistence solution. It generally relies on object-oriented database technology in order 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 [43]. 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.

7 Summary and Outlook

SqueakSave is a reflective object-relational mapper, that implements means to free 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.

Extensibility and configurability of the solution are viable factors for interoperability with other object-relational mapping solutions. We have presented an example application that can be utilized by two persistence frameworks with only slight adoptions. Additionally 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 state of implementation to some degree 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 [48] 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 during the course of this thesis, 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 functionality as an easily attachable aspect to existing applications [47]. The latter is possible by an in-depth analysis of inner-application workflows, that determine the queries most suitable within certain execution states [46].

Despite of the remaining potential for optimizations regarding the performance of the framework, SqueakSave provides a solid foundation for further research and has shown that meta-programming and reflection are viable means to simplify the integration of object-relational persistence mechanisms into applications that are developed within dynamically-typed object-oriented programming environments.

Bibliography

[1] A. Adya and J.A. Blakeley and S. Melnik and S. Muralidhar. Anatomy of the ADO.NET entity framework. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2007), ACM, pp. 877–888.

[2] Alashqur, A., Su, S., and Lam, H. OQL: a query language for manipulating object-oriented databases. In *VLDB '89: Proceedings of the 15th international conference on Very large data bases* (San Francisco, CA, USA, 1989), Morgan Kaufmann Publishers Inc., pp. 433–442.

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

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

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

[6] Barry, D., Berler, M., Eastman, J., Jordan, D., Schadow, O., and Velez, F. *The Object Data Standard: ODMG 3.0*. Morgan Kaufmann, 2000.

[7] Bayer, R., and Mccreight, E. Organization and maintenance of large ordered indexes. *Acta Znformatica 1*, 3 (1972), 173–189.

[8] Beck, K. *Smalltalk Best Practice Patterns*. Prentice Hall PTR, 1996.

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

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

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

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

[13] Codd, E. A relational model of data for large shared data banks. *Commun. ACM 13*, 6 (1970), 377–387.

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

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

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

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

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

[19] Fowler, M. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

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

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

[22] Goldberg, A., and Robson, D. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

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

[24] Groff, J., and Weinberg, P. *SQL: The Complete Reference*. Osborne/McGraw-Hill, 1999.

[25] Hallett, J. J., and Kfoury, A. J. A formal semantics for weak references. Tech. rep., Department of Computer Science, Boston University, 2005.

[26] Hirschfeld, R., Haupt, M., Berger, M. B. S., Eastman, J., Osburg, P., Perscheid, M., and Tibbe, D. *An Introduction to Seaside*, vol. 1. Software Architecture Group, Hasso-Plattner-Institute, 2008.

[27] Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., and Kay, A. Back to the future: the story of Squeak, a practical Smalltalk written in itself. *SIGPLAN Not. 32*, 10 (1997), 318–326.

[28] ISO/IEC. *SQL 9075-2008 standard*. ISO/IEC, 2008.

[29] Jarke, M., and Koch, J. Query Optimization in Database Systems. *ACM Comput. Surv. 16*, 2 (1984), 111–152.

[30] K. Barclay and J. Savage. *Groovy Programming: An Introduction for Java Developers*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[31] KIM, W. Research directions in object-oriented database systems. In *PODS '90: Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems* (New York, NY, USA, 1990), ACM, pp. 1–15.

[32] KLIMAS, E., THOMAS, D., AND SKUBLICS, S. *Smalltalk with style*. Prentice Hall, Englewood Cliffs, NJ, 1996.

[33] KNIGHT, A. 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)* (New York, NY, USA, 2000), ACM, pp. 173–174.

[34] LEAVITT, N. Whatever Happened to Object-Oriented Databases? *Computer 33*, 8 (2000), 16–19.

[35] LEE, S., KIM, S., AND KIM, W. The BORD Benchmark for Object-Relational Databases. In *DEXA '00: Proceedings of the 11th International Conference on Database and Expert Systems Applications* (London, UK, 2000), Springer-Verlag, pp. 6–20.

[36] LESER, U., AND NAUMANN, F. *Informationsintegration: Architekturen und Methoden zur Integration verteilter und heterogener Datenquellen*. Dpunkt Verlag, 2007.

[37] LIENHARD, A., AND RENGGLI, L. SqueakSource - Smart Monticello Repository. Tech. rep., Software Composition Group, University of Bern, Switzerland, June 2005.

[38] LODHI, F., AND GHAZALI, M. Design of a simple and effective object-to-relational mapping technique. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing* (New York, NY, USA, 2007), ACM, pp. 1445–1449.

[39] MARTIN, R. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.

[40] MELNIK, S., ADYA, A., AND BERNSTEIN, P. Compiling mappings to bridge applications and databases. *ACM Trans. Database Syst. 33*, 4 (2008), 1–50.

[41] MÜLLER, J., ENDERLEIN, S., HELMICH, M., KRÜGER, J., AND ZEIER, A. Customizing Enterprise Software as a Service Applications: Back-end Extension in a multi-tenancy Environment. In *Proceedings of the 11th International Conference on Enterprise Information Systems, Milan, Italy* (2009).

[42] OMG. UML 2.0 Specification, 2005.

[43] PATERSON, J., EDLICH, S., HÖRNING, H., AND HÖRNING, R. *The Definitive Guide to db4o*. Apress, Berkely, CA, USA, 2006.

[44] PETRI, C. A. *Kommunikation mit Automaten*. PhD thesis, Institut für instrumentelle Mathematik, Bonn, 1962.

[45] Pilato, M. *Version Control With Subversion*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2004.

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

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

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

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

[50] Stonebraker, M., and Moore, D. *Object Relational DBMSs: The Next Great Wave*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1995.

[51] SunMicrosystems. *MySQL 5.1 Reference Manual*. Sun Microsystems, Inc., 2009.

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

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

[54] Wiedermann, B., Ibrahim, A., and Cook, W. Interprocedural query extraction for transparent persistence. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications* (New York, NY, USA, 2008), ACM, pp. 19–36.

[55] Yoder, J., Johnson, R., and Wilson, Q. Connecting business objects to relational databases. In *Conference on the Pattern Languages of Programs* (St.Louis, Missouri, EUA, 1998).

[56] Zyl, P. V., Kourie, D., and Boake, A. 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* (Pretoria, Republic of South Africa, 2006), South African Institute for Computer Scientists and Information Technologists, pp. 1–11.