DynamicSchema: A Lightweight Persistency Framework for Context-Oriented Data Management

Sergio Castro, Sebastián González, Kim Mens
ICTEAM Institute
Université catholique de Louvain
Belgium
{sergio.castro, s.gonzalez, kim.mens}
@uclouvain.be

Marcus Denker
INRIA Lille
France
marcus.denker@inria.fr

ABSTRACT

While context-oriented programming technology so far has focused mostly on behavioral adaptation, context-oriented data management has received much less attention. In this paper we make a case for the problem of context-oriented data management, using a concrete example of a mobile application. We illustrate some of the issues involved and propose a lightweight persistency framework, called *DynamicSchema*, that resolves some of these issues. The solution consists in a flexible reification of the database schema, as a convenient dynamic data structure that can be adapted at execution time, according to sensed context changes. Implementing our mobile application using this framework enabled us to reduce the complexity of the domain modeling layer, to facilitate the production of code with low memory footprint, and to simplify the implementation of certain scenarios related to context-dependent security concerns.

Categories and Subject Descriptors

H.2.1 [**Information Systems**]: Logical Design—*data models*; H.2.4 [**Information Systems**]: Systems

General Terms

Design

Keywords

Context-Oriented Data Management, Data Access Layer, Dynamic Adaptability, Persistency

1. INTRODUCTION

This paper presents *DynamicSchema*, a lightweight persistency framework that aims at facilitating the implementation of context-dependent business rules governing the access to application data. In our framework, the data access layer is represented by a conveniently reified data structure. This reified model evolves according to detected context changes.

PREPRINT. Published at COP '12, Workshop, ECOOP 2012.
DOI: 10.1145/2307436.2307441

Although we do not claim that the proposed framework is necessarily the most adequate solution for any possible context-oriented data management scenario, we present a concrete example of the implementation of a mobile application in which the framework successfully helped us to

- decrease the complexity required to implement the domain modeling layer,
- facilitate the production of code with low memory requirements and other resources, and
- facilitate the implementation of certain security concerns, in the presence of dynamic adaptability.

The concrete case study of the context-aware mobile application is presented in Section 2. The case study serves as a basis to illustrate and discuss the merits of the proposed framework. Using some concrete examples from the case study, Section 3 illustrates some typical problems in context-aware applications when interacting with the data access layer. Section 4 then introduces the *DynamicSchema* framework. It explains the core ideas and illustrates how it can be used to solve the aforementioned problems encountered in the case study. Section 5 presents relevant related work and finally Section 6 presents our conclusions.

2. CASE STUDY: A MOBILE CITY GUIDE

The *MobileCity* guide is a touristic application implemented on the Android platform. The objective of this application is to help and inform tourists when visiting an unknown city.

2.1 The Data Model

Among others, the *MobileCity* application offers tourist information on various *points of interest* (POIs) in a city. Each POI has a (textual) description, an image, as well as some other attributes. As for a POI, each image has a textual description associated with it as well. Furthermore, POIs are organized hierarchically according to their location. For example, the POI *Brussels* is a child of the POI *Belgium*. The latter, being a root POI of the application does not have a parent POI.

Each user of the application has its own user profile, which contains among others the preferred language of the user and his age group. The age group can have only two possible values for now: *adult* and *child*. This profile influences the actual description that should be shown when browsing the

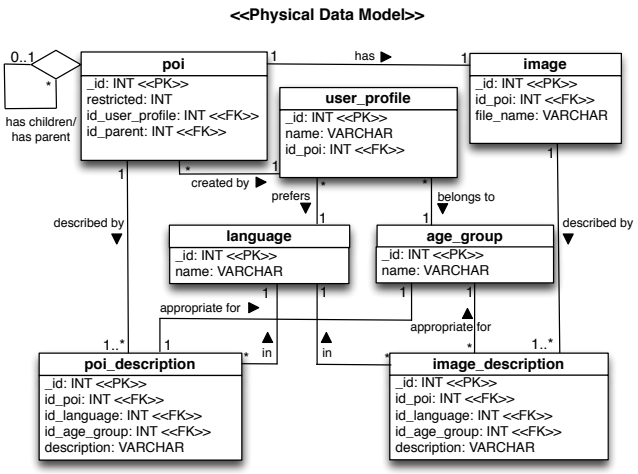

Figure 1: The *MobileCity* Physical Data Model.

information on POIs and images, so that the language and age group of the description match with those in the user profile. For example, if the user speaks *French* and is of the age group *child*, the application will use child-friendly descriptions in French for the POIs and images being shown.

All users have access to a list of predefined POIs (for example, encoded by the tourist office), but they can also add their own new POIs to the database. Newly defined POIs are accessible only to the user profiles under which they were created, since probably they are not of interest to other users of the application. Already existing POIs, on the other hand, are accessible to all users of the application and belong to a *Default* user profile.

Finally, there is an important security concern. Certain POIs can be marked as restricted (e.g. bars serving alcohol), which means that they should never appear when browsed by a user whose profile belongs to the *child* age group.

Figure 1 shows a partial physical data model [1] of our application. Please note that not all entities and attributes present in our actual implementation have been included, but only those needed to support the discussion in this paper.

2.2 The Object Model

Figure 2 shows a class diagram representing the object model of the application, corresponding to the data model of Figure 1.

Although a common rule of thumb when mapping a physical data model to an object model is trying to keep relation multiplicities the same [1], we have explicitly decided not to follow this convention for certain context-dependent relations present in our model. For example, in the class diagram of Figure 2, we have not considered the relation between a POI and its description to be a *one-to-many*, but rather a *one-to-one* relation. There are many reasons supporting this design decision. First, at any given time in our application, the only description that matters is the one corresponding to the current context or, to be more specific, to

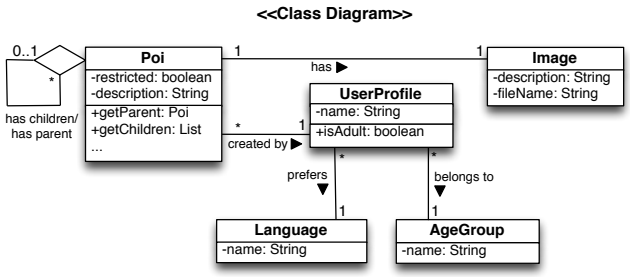

Figure 2: The *MobileCity* Class Diagram.

the profile of the currently active user (i.e. to the current user's language and age group). Modeling a POI as having only one description has the advantage that objects interacting with it do not have to be aware that in fact there are multiple descriptions, simplifying in this way the implementation. Second, this also contributes to keep the memory footprint low (which is an important requirement in mobile applications), given that a POI or image loaded in memory will contain only one description, instead of a list of all possible descriptions.

Given that we have decided to consider a POI description just as a piece of text, we added this description immediately as an attribute to the POI entity, instead of creating a separate entity as was necessary in the physical data model. The same remark applies to images and their description.

3. ISSUES WITH DATA MANAGEMENT IN CONTEXT-ORIENTED APPLICATIONS

In this section we take a look at some of the typical problems related to data management that arise in context-oriented applications similar to the one of our case study.

3.1 A simple scenario

Suppose we are interested in gathering all image entities from our database. According to the object model described above, we should get only one description per image, determined by the actual context. Also, for efficiency reasons, we would like to fetch the image descriptions from the database at the same time as when the image entities themselves are queried.

To implement these requirements, we implemented a variation of the *Table Gateway* pattern [2]. In our implementation of this pattern, dedicated objects encapsulate certain table-level operations such as queries.

```
1  public class ImageTableGateway extends TableGateway {
2    ...
3    public List<Image> getAll(UserProfile user) {
4      String idLanguage = user.getLanguage().getId();
5      String idAgeGroup = user.getAgeGroup().getId();
6      String query = "SELECT img._id, img.id_poi,
           img.file_name, imgd.description "
7        +"FROM image img "
8        +"LEFT JOIN image_description imgd ON img._id =
             imgd.id_poi_image "
9        +"AND imgd.id_language=? AND imgd.id_age_group=?";
10     Cursor cursor = db().rawQuery(query,
11       new String[] {idLanguage, idAgeGroup});
12     return adaptCursorToList(cursor, 0);
13   }
14 }
```

Listing 1: Querying all images entities, for a given

user profile

Listing 1 shows a partial implementation of the *ImageTableGateway* class. This class implements a method *getAll* (lines 3–13) that returns all images in the database. To do so, it first builds an SQL query (lines 6–9). This query collects the relevant data from the *image* and *image_description* tables (line 6). Note that this is a parameterized query, since the join condition of the *image_description* table depends on two parameters that will be bound later to the language and age group of the user profile entity sent as parameter (line 9).

Once the query has been built, it is evaluated to obtain a database cursor with its results (line 10) and transforms these results in a list of images. This final step is accomplished by invoking the auxiliary method *adaptCursorToList* (line 12) of which the implementation is not shown for brevity.

Although in general this can be regarded as an acceptable solution, there are certain issues with this implementation. For example, objects invoking this method must always send context information as a parameter. There are many possible patterns that can be applied to alleviate this issue, but before looking for a solution of this relatively simple problem, let us first consider a slightly more complex scenario.

3.2 Dealing with multiple context-dependant concerns

Let us now focus on the POI entities. As in the previous example, we require that whenever a POI entity is retrieved from the database, its description is also retrieved. In addition, in our *MobileCity* application, when retrieving a POI we should also retrieve its related image and that image's description.

Let us suppose that we are interested in finding all root POIs.

```
1  public class PoiTableGateway extends TableGateway {
2    ...
3    public List<Poi> getRootPois(UserProfile user) {
4      String idLanguage = user.getLanguage().getId();
5      String idAgeGroup = user.getAgeGroup().getId();
6      String query = "SELECT p._id, p.parent,
           p.id_user_profile, p.restricted, pd.description,
           "
7        +"img._id, img.file_name, img.id_poi,
             imgd.description, "
8        +"FROM poi p "
9        +"LEFT JOIN poi_description pd ON p._id = pd.id_poi
             "
10       +"AND pd.id_language = ? AND pd.id_age_group = ? "
11       +"LEFT JOIN image img ON p._id = img.id_poi "
12       +"LEFT JOIN image_description imgd ON img._id =
             imgd.id_poi_image "
13       +"AND imgd.id_language = ? AND imgd.id_age_group =
             ? "
14       +"JOIN user_profile up ON p.id_user_profile =
             up._id "
15       +"WHERE p.id_parent IS NULL "
16       +"AND (p.id_user_profile = ? OR up.name='Default')
             ";
17     if(!user.isAdult())
18       query += " AND p.restricted = 0";
19     Cursor cursor = db().rawQuery(query,
20       new String[] {idLanguage, idAgeGroup,
21       idLanguage, idAgeGroup, user.getId()});
22     return adaptCursorToList(cursor, 0);
23   }
24 }
```

Listing 2: Querying all root POI entities, for a given user profile

As in the previous example, we provide a *PoiTableGateway* class (listing 2) that implements some methods to query the POI table. In this class, the method *getRootPois* returns a list with all the root POIs in the database. The structure of this method is very similar to the *getAll* method in the *ImageTableGateway* class. First a query is built (lines 6–18). The relevant data is retrieved from the tables *poi*, *poi_description*, *image* and *image_description* (lines 6–7). We then specify that the join condition with the *poi_description* table depends on two query parameters. These parameters will be bound later to the language and age group of the current user (line 10). Until here the example is very similar to the previous one.

But we also need to add an additional join condition to retrieve the right image description, for the image corresponding to the retrieved POI and according to the language and age group of the current user profile (lines 12–13). To filter the results we add a *WHERE* clause to include only POIs that do not have a parent (line 15). In addition, according to our requirements we should also filter the POIs themselves according to the active user profile (we need to collect all POIs in the *Default* user profile plus all the POIs stored in the current user's own profile). We do so by adding to the *WHERE* clause a condition filtering the POIs according to their user creator (line 16) and to the age group of the current user (lines 17–18).

3.3 Discussion

As in the previous example, callers of the *getRootPois* method should pass context-specific data as a parameter to the method (i.e., the currently active user profile). In addition to that issue, this example has shown an increasing complexity with respect to the first one. Part of the reason for this is the need to be aware of how context changes influence not only the direct relations of a table being queried, but also all transitive relations that participate in the query. If many such relations are present, this produces data access code that quickly becomes difficult to maintain and evolve.

In addition to the complexity of context-dependent relations, we also need to be aware of invariant constraints (e.g., security concerns) that should remain respected in any kind of access to an entity. To illustrate this additional problem, let us recall that in our last example we were querying POI entities and that we explicitly added filters for limiting the access to certain POIs (for children). These filters should be present not only in all direct queries to the POI table, but also whenever querying an entity with a direct or transitive relation with a POI entity.

These additional constraints that need to be added everywhere, quickly lead to heavy code duplication and again to data access code that becomes very difficult to maintain and evolve. To overcome these issues, in the next section we present a solution to some of the problems discussed in this section.

4. THE *DYNAMICSCHEMA* FRAMEWORK

Part of the problems shown in section 3 are related to the difficulty in modularizing the application data structure, their data constraints, and their context depending rules. Although some frameworks can help to alleviate some of these

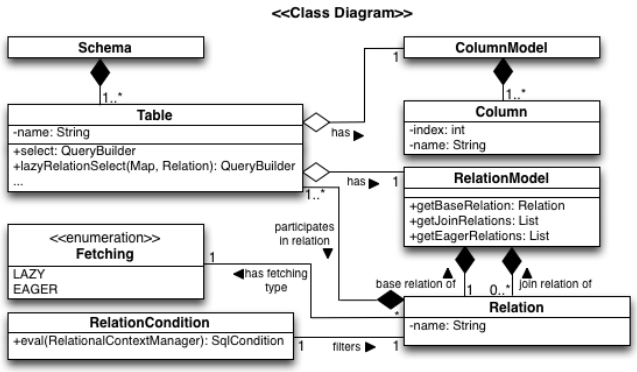

Figure 3: *DynamicSchema* Class Diagram.

problems (see section 5), they do not provide facilities to deal with complex context depending rules.

DynamicSchema alleviates this problem. A database schema is modeled as a dynamic structure that can mutate according to changes in a context. Using some prepackaged routines, applications use this structure to generate required SQL code, so changes on this structure will immediate affect the way data is accessed and manipulated.

4.1 The framework architecture

Before showing how *DynamicSchema* can be used to represent schema objects, we introduce its core classes. Figure 3 shows its class diagram. The *Schema* class is a repository of database schema objects (limited to table objects only for now). Table objects, in addition to a name, have a column model and a relation model. A column model models the columns in a table. The relation model defines a set of relations with other tables. We refer to the table defining the relation model as the *base table*, to distinguish it from other tables participating in relations with it. Each relation has: a name, the list of tables participating in the relation, a default fetching mode and a relation (join) condition. The fetching mode can take two possible values: *EAGER* or *LAZY*. *EAGER* fetching means that the relation has to be taken into account when generating a query on the base table. *LAZY* fetching means that the relation should be ignored when querying the base table, since entities in the relation tables will be brought lazily later when needed.

The relation condition describes how to match entities in the base table with entities in the relation tables (i.e., a join condition). When a relation condition is specified in the base relation, it will work as a filter that will be applied to all the results of a query involving this table (translated to SQL as a *WHERE* condition).

4.2 Modeling a database schema

Listing 3 shows how the *poi* table can be modeled using our framework. On lines 4–7 we define singleton objects that help us to refer to the table name, its column and relation model. The *PoiColumns* class (lines 9–18) defines the column model of the table.

```
1  import static android.provider.BaseColumns._ID;
2  ...
3  public class PoiTable extends Table<PoiColumns,
       PoiRelations> {
4    public static final String NAME = "poi";
5    public static PoiTable table = new PoiTable();
6    public static PoiColumns columns() {return
         table.getColumnModel();};
7    public static PoiRelations relations() {return
         table.getRelationModel();}
8
9    public static class PoiColumns extends ColumnModel {
10     public Column ID = new Column(_ID);
11     public Column ID_PARENT_POI = new Column("parent");
12     public Column ID_USER_PROFILE = new
           Column("id_user_profile");
13     public Column RESTRICTED = new Column("restricted");
14     ...
15     public PoiColumns() {
16       setColumns(ID, ID_PARENT_POI, ID_USER_PROFILE,
             RESTRICTED, ...);
17     }
18   }
19   ...
20   public static class PoiRelations extends RelationModel {
21     public Relation childrenPoi;
22     public UserDependingRelation poiDescription;
23     public Relation poiImage;
24     public Relation userProfile;
25     ...
26     public PoiRelations() {
27       childrenPoi = new Relation(this,
             CHILDREN_POI_RELATION_NAME, Fetching.LAZY,
             PoiTable.table);
28       childrenPoi.setCondition(new RelationCondition() {
29         @Override public SqlCondition
               eval(IRelationalContextManager ctx) {
30           return new
                 SqlCondition().eq(baseRelation.col(ctx,
                 PoiTable.columns().ID),
                 childrenPoi.col(ctx,
                 PoiTable.columns().ID_PARENT_POI));
31         }
32       });
33       poiDescription = new UserDependingRelation(this,
           POI_DESCRIPTION_RELATION_NAME,
           PoiDescriptionTable.table);
34       poiDescription.setCondition(new RelationCondition()
             {
35         @Override public SqlCondition
               eval(IRelationalContextManager ctx) {
36           Integer languageId =
                 poiDescription.getUserProfile().getIdLanguage();
37           Integer ageGroupId =
                 poiDescription.getUserProfile().getIdAgeGroup();
38           SqlCondition condition = new
                 SqlCondition().eq(baseRelation.col(ctx,
                 PoiTable.columns().ID),
                 poiDescription.col(ctx,
                 PoiDescriptionTable.columns().ID_POI))
39             .and().eq(poiDescription.col(ctx,
                   PoiDescriptionTable.columns().ID_LANGUAGE),
                   languageId)
40             .and().eq(poiDescription.col(ctx,
                   PoiDescriptionTable.columns().ID_AGE_GROUP),
                   ageGroupId);
41
42           return condition;
43         }
44       });
45       poiImage = ...
46       userProfile = ...
47       ...
48       setJoinRelations(childrenPoi, poiDescription,
             poiImage, userProfile);
49     }
50   }
51 }
```

Listing 3: Modeling the *poi* table

The relation model is given by the class *PoiRelations* (lines 20–50). The 4 relations of the *poi* table are defined in this model: *childrenPoi* (lines 27–32), *poiDescription* (lines 33–44), *poiImage* (line 45) and *userProfile* (line 46). From these relations, only *childrenPoi* is a lazy relation (note the *LAZY* constant in the initialization of the relation on line 27). Then, entities from the other tables participating in the other three relations will be brought from the database each time a *poi* entity is queried (if not specified otherwise, relations are eager in our framework). Those related entities are defined by the relation conditions. For example, the *childrenPoi* recursive relation indicates that a POI can have a list of children POI. In plain SQL, the query consulting all the children of a POI with a certain *id* is shown below:

```
SELECT p2.* FROM poi p1 JOIN poi p2 ON p1._id =
    p2.parent WHERE p1._id = ?
```

In this query, the *poi* table appears twice since the relation is recursive. Then, we need a mechanism to specify which role of the relation the *poi* table is playing. In SQL this is accomplished with the use of distinct aliases (e.g., *p1* and *p2*).

The same join condition shown in the previous SQL example is specified in our framework at line 30 of listing 3. In this line, the expression:

```
baseRelation.col(ctx, PoiTable.columns().ID)
```

can be read as: The *ID* column at the *poi* table in the context of the base relation.

In a similar way, the expression:

```
childrenPoi.col(ctx,
    PoiTable.columns().ID_PARENT_POI)
```

can be read as: The *PARENT_POI* column at the *poi* table in the context of the *childrenPoi* relation.

Now let's focus on the *poiDescription* relation (lines 33 to 44). Note that this relation is not an instance of the *Relation* class as all the other relations in the relation model. Instead, it is an instance of *UserDependingRelation*, which extends *Relation* with the addition of a *UserProfile* instance variable. This class is not part of our framework, but has been added as part of the *MobileCity* application to model relations which join conditions depend on certain values of the current user profile. In the *poiDescription* relation, we can see that the join condition depends on the preferred user profile language (line 39) and its age group (line 40).

4.3 SQL generative routines

This section provides an intuition of how SQL code is generated from our database model.

In general, our framework navigates and manipulates schema objects through the use of visitors. There are two different kind of visitors pre-packaged in the framework: The *TableRelationsVisitor* and the *SchemaVisitor*.

The *TableRelationsVisitor* visits all the relations, and tables in those relations, of a given base table. The *SchemaVisitor* visits all the tables and their relations in a given schema.

Extending these generic visitors, we can easily accomplish operations such as generating SQL statements, or modifying certain objects in the schema according to sensed context changes.

The *SelectBuilderEagerRelationsVisitor* class is an example of how extending *TableRelationsVisitor* is possible to generate SQL statements. A fragment of this class is shown in list 4. The relation context object passed by at the constructor (line 2) has the knowledge of how tables and column names should be resolved. It basically acts as a SQL alias manager.

Without entering into technical details, the two *doVisit* methods (lines 7–13 and 15–18) create a query manipulating a query builder object.

```
1  public class SelectBuilderEagerRelationsVisitor extends
       TableRelationsVisitor {
2    protected IRelationalContextManager ctx;
3    private ContextedQueryBuilder queryBuilder;
4
5    public
       SelectBuilderEagerRelationsVisitor(IRelationalContextManager
       ctx) {...}
6    ...
7    @Override public boolean doVisit(Relation relation) {
8      ...
9      //adding relation join conditions to the query
10     LeftJoin join = new
         LeftJoin(relation.joinCondition(ctx),
         getTableAliases(relation));
11     queryBuilder.addJoin(join);
12     ...
13   }
14
15   @Override public boolean doVisit(Relation relation,
       Table table) {
16     //adding columns to the query and table specific
         filter conditions
17     ...
18   }
19 }
```

Listing 4: Generating a query

4.4 *DynamicSchema* in action

4.4.1 Adapting the schema to context changes

In *DinamicSchema*, the structures reifying a database schema are not static, but can vary according to changes in the context.

When the context changes, these schema modifications can be easily implemented with a class extending the *SchemaVisitor* visitor described in section 4.3.

A visitor updating the schema of the *MobileCity* application is shown in listing 5. This class overrides the *doVisit* method (lines 6 to 29) inherited from the *SchemaVisitor* class. For each visited relation it checks if the relation is an instance of *UserDependingRelation* (line 9). If that is the case it will set a new user profile (line 10).

If the currently visited relation is in fact the *PoiTable* base relation (line 11), it modifies the relation filtering conditions. For example, a condition filtering restricted POIs (line 21) is only added after verifying that the current user profile is not the one of an adult (line 20).

```
1  public class ContextChangedVisitor extends SchemaVisitor {
2    private PropertyChangeEvent changeEvent;
3    private UserProfile defaultUserProfile;
4    public ContextChangedVisitor(PropertyChangeEvent
       changeEvent, AbstractDataSource dataSource) {...}
5
6    @Override public boolean doVisit(final Relation
       relation) {
7      if(changeEvent.getPropertyName().equals(USER_PROFILE_CONTEXT_CHAN
         {
8        final UserProfile newUserProfile =
           (UserProfile)changeEvent.getNewValue();
9        if(relation instanceof UserDependingRelation)
10         ((UserDependingRelation)relation).setUserProfile(newUserProf
11       if(relation.equals(PoiTable.relations().getBaseRelation()))
           {
12         final int restricted =
             newUserProfile.isAdult()?0:1;
13         RelationCondition condition = new
             RelationCondition() {
14           @Override public SqlCondition
               eval(IRelationalContextManager ctx) {
15             SqlCondition sqlCondition = new
                 SqlCondition();
16             sqlCondition.group( //parenthesed expression
17               new SqlCondition().eq(relation.col(ctx,
                   PoiTable.columns().ID_USER_PROFILE),
                   newUserProfile.getId())
18               .or().eq(relation.col(ctx,
                   PoiTable.columns().ID_USER_PROFILE),
                   defaultUserProfile.getId())
```

```
19              );
20              if(!newUserProfile.isAdult())
21                sqlCondition.and().not().eq(relation.col(ctx,
                      PoiTable.columns().RESTRICTED),
                      restricted);
22              return sqlCondition;
23            }
24          };
25          relation.setCondition(condition);
26        }
27      }
28      return true;
29    }
30  }
```

Listing 5: Adapting the schema to context changes

4.4.2 *Querying the schema*

The framework provides convenient methods to generate queries. These methods are implemented by the *Table* class.

Listing 6 shows an example of one of them: the *select* method (lines 3 to 8). This method answers an object representing a SQL query. This query is generated with the *SelectBuilderEagerRelationsVisitor* visitor (line 6) described in section 4.3. As mentioned before, this visitor generates a query that will bring, when executed, all the entities persisted on the table. In addition, entities in other tables participating in eager relations with the queried table, are also included in the query.

In the generated query, join conditions and other filters are resolved depending on the current state of the relation models of all the *Table* objects involved.

```
1  public abstract class Table<ColumnModelType extends
       ColumnModel, RelationModelType extends
       RelationModel> {
2    ...
3    public ContextedQueryBuilder select() {
4      IRelationalContextManager relationalContext =
           getRelationalContext();
5      SelectBuilderEagerRelationsVisitor
           selectBuilderVisitor = new
           SelectBuilderEagerRelationsVisitor(relationalContext);
6      selectBuilderVisitor.visit(this);
7      return selectBuilderVisitor.getQueryBuilder();
8    }
9    //other query methods ...
10 }
```

Listing 6: The *Table* class

In addition to the filter conditions generated by the *select* method, it is possible to include new filters if necessary.

As an example, listing 7 shows an alternative version of the *getRootPois* method shown in listing 2. If we compare it with the previous implementation, we observe that this method is not parameterized with context information. Also, a query bringing *all* the POIs from the database is generated with only one line (line 4). We just add a filter condition checking that only POIs with no parents are included (line 6) and all the work for building the query has been done. The rest of the method is similar to the previous version.

```
1  public class PoiTableGateway extends TableGateway {
2    ...
3    public List<IPoi> getRootPois() {
4      ContextedQueryBuilder queryBuilder =
           PoiTable.table.select();
5      IRelationalContextManager ctx =
           queryBuilder.getRelationalContext();
6      queryBuilder.addWhere(PoiTable.table.getBaseRelation().col(ctx,
           PoiTable.columns().ID_PARENT_POI) + " IS NULL");
7      Cursor cursor = rawQuery(queryBuilder);
8      return adaptCursorToList(cursor, 0, ctx);
9    }
10 }
```

Listing 7: An improved implementation of the *PoiTableGateway* class

This example has shown how, with our framework, we can avoid to write boilerplate code related to data access, reduce the need to parameterize persistency methods with context information, and centralize data access rules such as context depending security concerns.

5. RELATED WORK

Most popular persistency frameworks cannot be easily used in mobile programming. In the Java world (then in the Android world), this is both for the relatively big amount of resources they require, or other technical problems such as a dependency on JDBC (e.g., Hibernate [4]).

There are lightweight persistency frameworks that can work in mobile devices (e.g., OrmLite [6]) , or are specifically designed for them (e.g., greenDAO [3], ActiveAndroid [5]). However, these frameworks are based on the use of annotations or configuration files to map model classes to database tables (e.g., OrmLite, ActiveAndroid), or in code generation techniques to produce once the code that will interact with the database (e.g., greenDAO). Therefore, they consider the database as a static structure where relations are fixed and do not depend on a context.

6. CONCLUSIONS

In this paper we have illustrated some of the complexity related to context-oriented data management.

As discussed in section 1, part of this complexity is introduced by the presence of context depending mappings between the physical model and the class model. In our case study, the need to use these context depending relations is a direct consequence of the desire of keeping the model simple, minimizing its memory requirements and consistent all the time with the security concerns of the application.

We have shown how with a dynamic structure reifying a database schema, we could simplify the programming of the data access layer implementing these requirements. Furthermore, we show how easily we can update our database schema representation in the case of context changes.

7. REFERENCES

[1] S. W. Ambler. *Agile Database Techniques*. Wiley Publishing, third edition edition, 2003.

[2] M. Fowler. *Patterns of Enterprise Application Architecture*. Pearson Education, 2003.

[3] greenrobot. greendao. http://greendao-orm.com/.

[4] G. King. Hibernate. http://www.hibernate.org/.

[5] M. Pardo. Activeandroid. https://www.activeandroid.com/.

[6] G. Watson. Ormlite. http://ormlite.com/.