Analysis and Implementation of Database Concerns Using Aspects

Murat Kurtcephe, Oğuzcan Oğuz and Mehmet Özçelik

Department Of Computer Engineering, Bilkent University
06800 Bilkent, Ankara, Turkey
{kurtcephe,oguzcan,,ozcelik} @ cs.bilkent.edu.tr

Abstract—*Data intensive applications have to deal with different concerns which cannot be easily modularized using conventional abstraction techniques, such as OOSD. In this paper we will discuss the implementation of an example database system on the case study, Car Rental System, using an aspect-oriented approach. We will focus on the concerns persistence, synchronization, failure management and refresh from database. We will report on our lessons learned as performance issues and hardship of finding the pointcuts for the aspects.*

***Keywords*—Aspect-Oriented Software Development, database, persistence**

1. Introduction

Aspect-Oriented Software Development (AOSD) mainly works on the crosscutting concerns to decrease the scattering and tangling codes. Also, it tries to increase the modularity of the system for reuse. Persistence is an already faced issue where some aspect-oriented solutions have been provided before. For an existing system, without any changes in the object-oriented design and without the knowledge of application classes about persistence, we can implement a database system to the application by aspects. Other than persistence, we dealt with synchronization, failure management and refresh from database issues. Synchronization concern focuses on the authorization of a modification operation on an object. Failure management concern deals with the database failures in a simple way. Last concern is refreshing the objects with the new data in database. This concern is a result of multi-user application. As we use the database as a way of communication between different users.

After we determined the concerns to overcome during the implementation of the database system, we decided that aspect-oriented approach would be better choice as we are dealing with a legacy code. For each concerns, we needed to change the existing code for database operations. Also, for synchronization and failure management concerns we needed to access to and change user interface codes, which is not preferable as it permits scattering codes. To maintain molarity and decrease the scattering codes, we preferred aspect-oriented approach.

Our solution includes separation of concerns for the database aspects. We divided persistence aspect in four parts. These parts consist of initialization, addition, update and deletion operations. Other concerns that we face during the implementation are synchronization, failure management and refresh from database. Synchronization aspect focuses on the modification operations. It uses a lock for the modified object and controls lock before entering the modification operation. Failure management aspect checks whether the database connection is valid before every query execution. Lastly, refresh aspect updates the data of the object each time a method of the object is called.

In [1], it is stated that persistence concern can be implemented using AOSD to be reused in other applications which does not have a database system. However, for our case we face with issues where the difference between a temporary object and a stored object is not clear. For example, an update of an object requires both old and new objects. Therefore, a reusable aspect could not differentiate whether an object is database related or not. This brings forward the issue that some projects could require implementing aspects for specific parts of the code. In our case, we faced with this problem and realized our implementation according to that.

We applied a database management system to an existing code by using aspect-oriented programming. This allowed us to increase modularity and decrease the possibility of scattering codes. We also focused on synchronization and failure management to make database management system stable. However, even if aspect-oriented approach helped us implementing some of the concerns, there were some issues which should not be ignored. We faced with performance issues in refresh from database concern and update part of persistence concern. For the first one – refresh

from database – we lacked performance but the implementation was easy. On the other hand, we implemented update part of the persistence concern in a way to increase performance which brought harder implementation. Other problem we faced was hardness to find the necessary pointcuts. Aspect-oriented approach in synchronization, refresh from database and update concerns brought this issue.

In the following, section 2 provides the overview about our case study Car Rental System. Section 3 describes the implementation methodology we use for the aspects. Section 4 focuses on the discussion. Section 5 consists of related work and in the last part there is the conclusion.

2. Case Study: Car Rental System

Our case study is Car Rental System, which is implemented in object-oriented manner. As it can be derived from the name, it is an application allows renting and reserving a car from a branch. In fact design allows many users and many branches but there was no network implementation so it forced us to use it as a single user application. However, with the database integration and refresh aspect we can use the application on different machines in the same time as an online system as the difference can be seen in Fig 1.

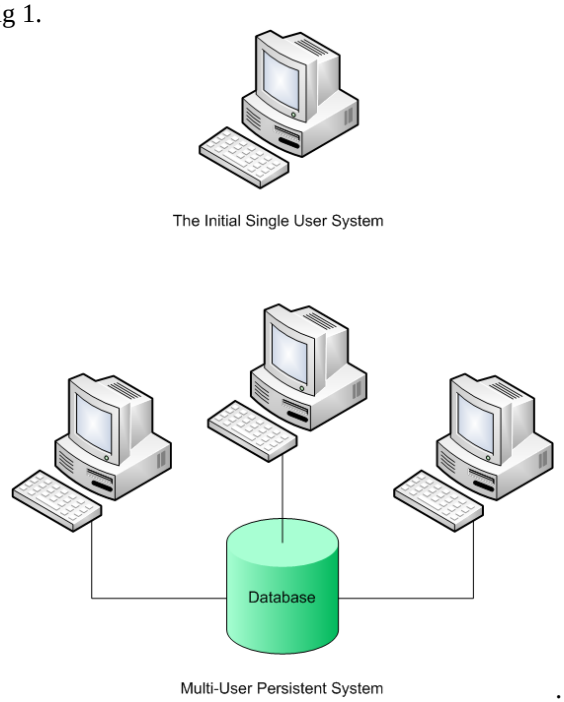

Fig. 1 The View of System After Implementing Aspects

In Car Rental System salesperson, local manager and main manager can login to the application. Before the aspect implementation, we were reading the data from a file which holds the information about the branches, customers, salesperson etc. However, with the aspect implementation, we ignore the file and use initialization aspect to create the objects using the database. After a salesperson logins to system, s/he can add&modify a customer, make rents and reservations for the already added cars and customers. The application enables salesperson to change the time and the car of the existing reservations and rents. As mentioned other than salesperson local manager and main manager can login to system. A local manager, addition to the capabilities as a salesperson, can add&modify a car description -as cars with same model, brand, year and color can be found, the application uses car descriptions where each car has a car description-, can add&modify a car, can add&modify a salesperson and can modify the information about the branch where he manages. All the additions and modifications made by a local manager is limited to the branch s/he manages. A main manager, addition to the capabilities as an local manager, can add new branches to the system and has the authority over all the branches.

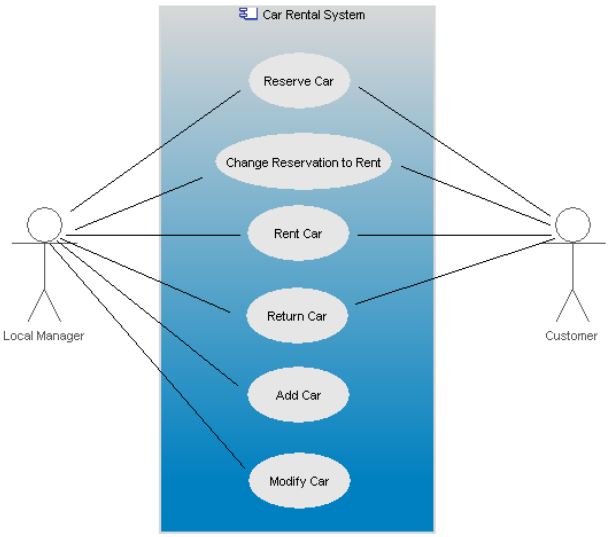

Fig. 2 Use Case Diagram of Local Manager and Customer

To pass a database system we firstly created the tables by using the attributes of the classes to be stored. Fig 3 shows a partial domain model for car rental system which corresponds to the ER diagram in Fig 4.

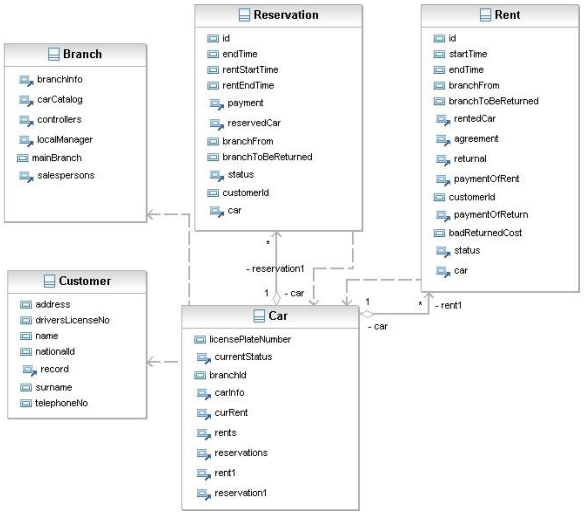

Fig.3 Partial UML Domain Model

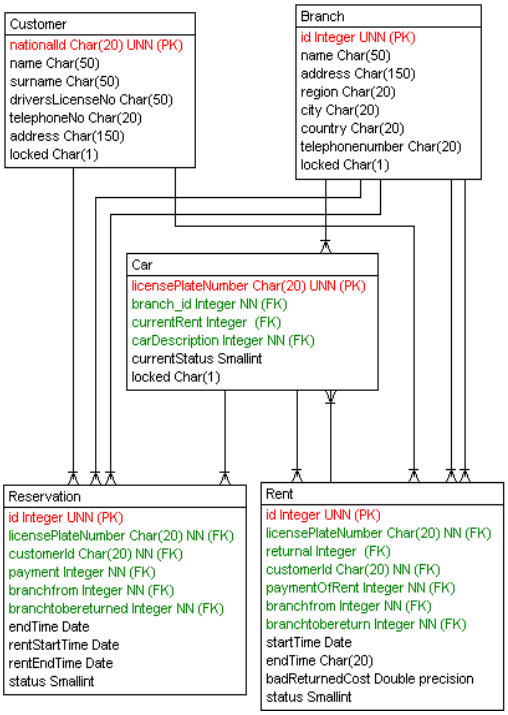

Fig.4 Partial ER diagram of the car rental system

Patterns Used in the Car Rental System

1. Abstraction-Occurrence

For the pattern we have a good example where we hold the descriptions of the cars seperated from the cars themselves. Since the description which holds brand, model, color and year of a car which could represent more than one car. Therefore we don't keep same data for different cars in themselves but in a seperated object called car description. To distinguish cars from each other we use license plate number. Each car has a unique license plate number and a car description which could be shared with other cars.

2. Delegation

As we used classes like Salesperson and LocalManager to reach the functions of Branch and MainBranch class, where the implementation of the functions are not done in both classes but in the class where all the information are hold like CarCatalog. CarCatalog has a function getCars which takes some arguments to specify the cars to return. This method is called by the function of Branch class getCars. It directly calls the function of CarCatalog not to implement the same thing. After that Salesperson reaches the needed implementation by calling the getCars function of Branch.

3. Player-Role

We have reservation, rent and car which have different status depending on the supplied situation. If a car has an accident then car has status "In Maintenance", and if it is rented it has "Rented" as the status and for each status car has it gives different responses to the callee functions. For this purpose we separated the roles,status, from car and used car status as a way to describe the current role of the car.

4. Façade

For the user interface to achieve the needed methods of the each classes we don't create an object for that class and call its functions but have class Salesperson which is some kind of interface between user interface and core classes. User interface has an instance of Salesperson and which has needed functions for GUI and this functions implements or calls the functions of other core classes.

3. Approach For Implementing Database Concerns With AOP

Applications using databases to store persistent data would have database related concerns such as persistency, synchronization, refreshment, and management of database or connection related failures. These concerns need to be addressed by most of the modules of the application and would be scattered through the application code. For this reason, these concerns cannot be perfectly modularized using solely object oriented design; an aspect oriented view is required.

1) Persistence

Persistency concern is composed of four set of operations: initialization operations, store operations, update operations, and deletion operations.

1.1) Initialization

Persistent storage for an application ensures that the application data can be stored and retrieved on demand. The application could be resumed at any time following a crash, a turn off etc. Between the cut point and resume point of an application, the persistent data may be changed by other possible applications that use the same storage. While resuming the application, the stored states and data need to be retrieved from the persistent storage and the application needs to be initialized with the retrieved data appropriately. Since the majority of the database systems are of relational databases with %80 of the market share, a relational database is used in this study. The systems that were not developed with database functionality in mind have their object models designed according to application functionality and concerns. These object models need to be transformed into relational tables to be stored in a relational database system. The resulting entity relationship model needs to be capable of storing all the required states and data to initialize the intended application. (see Fig 3 and Fig 4)

```
1 public aspect Initialization{
2
3   pointcut initializing(): execution(public static void CarRentalSystem.main(..));
4   before(): initializing()
5   {
6     //Initialize: Retrieve persistent data & Instantiate the objects accordingly
7   }
8
9   Branch initBranch(...){...}
10  Car initCar(...){...}
11  Customer initCustomer(...){...}
12
13  ...
14
15 }
```

Fig. 5 Initialization Aspect

To initialize an application with the persistent data, one would retrieve the states and data and then create instances of the objects which the application requires to run, in the memory. We propose an initialization aspect to perform initialization operations upon the start of the application. A refresh aspect that is later to be explained, cannot substitute the initializing aspect since at a point during runtime, there can be some objects that do not need to be refreshed but required to perform operations. For this reason, an initialization aspect is mandatory if we do not want to modify the existing code and modularize the initialization operations. Once the object model residing in the main memory is created, the initialization aspect needs to provide handles to the control of the application in order to ensure that the object instances can be achieved at required points. As we did implement the proposed aspects in Java, automatic garbage collection ensures that the instantiated objects can be reached provided that their handles are valid.

In the initialization aspect, shown in Fig. 5, one would benefit modularizing instantiate object operations. For every object to be initialized, there should be a initObjectName() method (see Fig 5, lines 9-11). These methods are given parameters required to specify the intended object in the database and possibly handles to other previously created objects, and the methods returns handle of the instantiated object. These methods would have high re-usability. The aspect to retrieve data from the database, for instance, would require initialization methods whenever retrieval for instantiation of an object is to be done. In fact, the mentioned methods that are specific to each object in the system can be placed in a separate aspect.

Order of instantiation of the objects is important since some objects would need handles to others and revisiting the objects would make the process more complex. In our example system, a top to bottom order is followed to minimize revisiting. The retrieval of data and instantiation of the objects could be performed in a recursive manner. Recursively instantiating objects would mean that once the initialization is started, an object is retrieved form the database and instantiated whenever it is required. In order not to fetch and instantiate same objects from the database for more than once, a caching mechanism could be employed such that handles to all previously instantiated objects would be stored in data structure in case same objects would be needed again.

During the initialization, the methods that belong to objects, such as set methods, and constructors are highly used to instantiate and set the states of the objects properly. While calling constructors and set methods, special care needs to be taken in order not to trigger other aspects that are waiting these calls to perform store or update operations.

1.2) Add methods

This concern focuses on the addition of the objects. It deals with the insertion queries of the database.

We can not easily modularize this concern by using OOSD because it needs to capture the points where addition operations are done. We have an option to change the legacy code by adding insertion queries where addition operations are done but to modularize the concern using aspect-oriented approach is preferable.

In this application, most of the objects (car, branch etc.) are held in array lists and we use this information

for our advantage. In the first part of the aspect, we get the add operations on the arrays and insert the added object to the database checking the instance of the object. In the second part, other than array list, hash tables are used to hold the data such as customer and salesperson. For these objects, as Fig. 6 shows, we get the put methods and after checking the instance type of the added object, we insert it to the database.

```
1  pointcut putMethods( Branch branch, Object addedObject ) :
2      call( * *.put( Object, Object ) ) && this( branch )
3      && args( Object, addedObject ) && !withincode( * *.get*(..) )
4      && within( core.* );
5
6  after( Branch branch, Object addedObject ) : putMethods ( branch, addedObject ) {
7      if ( addedObject instanceof Salesperson ) {
8          ...
9      }
10     else if ( addedObject instanceof Customer ) {
11         ...
12     }
13     else if ( addedObject instanceof Controller ) {
14         ...
15     }
16 }
```

Fig. 6 Pointcut and advice for addition of the objects held in hash tables

An important point which should not be ignore is that, there are some objects which refers other objects such as a rent holds a return object and two payment objects. Therefore, before inserting the rent object we insert the return and payment objects to the database.

Aspect-oriented approach minimizes the code scattering between the core classes and modularizes the addition concern in an aspect.

1.3) Update methods

Changing and modifying the objects of a program is another important issue about the persistency. Each object can be updated in a place of the execution. Therefore, the update operations in database is a cross cutting concern.

In OOP approach this could be handled by coding the update operations of database inside the code. By doing this, code starts to get tangled and it will be hard to maintain the code.

If we look the problem from AOP perspective then we will see that it is not easy to find the pointcuts where an object is update. Our case study was coded in Java, it is programming rule to declare getters and setters for each variable, and we can assume that each set operation is an update on the database. It could be very easy to implement such an approach. But when we do some search on the code we saw that each operation for example to modify a car description calls five set operations and this means the same object updated four times for nothing. It is a very simple example for our case if we think a larger object which has more that thirty features we can see the performance losing.

```
1  pointcut updateSalesperson( String oldSalespersonId
2  , Salesperson newSalesperson)
3  : execution(* Branch.updateSalesperson( String , Salesperson ))
4  && args(oldSalespersonId, newSalesperson);
5
6  pointcut setStartTimeRentOperations( Date changedTime )
7  :(execution(* Rent.setStartTime(Date)) ||
8  execution(* Reservation.setRentStartTime(Date)))
9  && args(changedTime);
10 pointcut setEndTimeRentOperations( Date changedTime )
11 :(execution(* Rent.setEndTime(Date)) ||
12 execution(* Reservation.setRentEndTime(Date))) && args(changedTime);
13 pointcut setEndTimeReservationOperation( Date changedTime )
14 :execution(* Reservation.setEndTime(Date))
15 && args(changedTime);
16 pointcut setCarStatus(CarStatus newStatus,Car target)
17 :execution(* Car.setStatus(CarStatus)) &&
18 args(newStatus) && this(target);
19 pointcut setCurrentRent(Rent rent,Car target)
20 :execution(* Car.setCurRent(Rent)) && args(rent) && this(target);
21 pointcut setPaymentAmount( double amount, Payment target )
22 :execution(* Payment.setAmount(double))
23 && args(amount) && this(target);
24 pointcut setPayment( Payment payment, Object target )
25 :call(* *.setPayment*(Payment)) && args(payment)
26 && target(target) && within(gui.*);
27 pointcut setRentStatus( RentStatus status, Rent target )
28 :execution(* *.setStatus(RentStatus))
29 && args(status) && this(target);
30 pointcut setCustomerId( String id, Object target )
31 :execution(* *.setCustomerId(String)) && args(id) && this(target);
32 pointcut setBranchToBeReturned( int branchId, Object target )
33 :execution(* *.setBranchToBeReturned(int))
34 && args(branchId) && this(target);
```

Fig 7: Some of the update operations pointcuts

Due to the performance considerations we choose another way for deciding join points. In this approach we try to define the operations which are updating the whole object the line 1 in figure shows a pointcut which is for catching the operation which updates the sales person. If each operation was working like that then we could be able to define a more reusable pointcuts. Rent operations are updating special fields of the rent (as shown in the sixth line of Fig7), such as start time of a rent. Then our first approach will fail because it is a specific field of the object, not the whole object. Therefore, we had to implement both pointcuts which are updating the whole objects, and the specific fields when needed.

After implementing the update concern, we saw that the aspect code we had is not very reusable and it is case specific. Also it is hard maintain such a code when an application has more than hundred different objects. It will be very hard to mine the aspects from the legacy code because of the reasons mentioned before. This approach (taking the operations as pointcuts) is not suitable for managing and re-usability. But on the other hand, taking the set operations as pointcuts decreases the performance of the program. There is a trade-off between performance and code re-usability.

1.4) Delete methods

In our case, the car rental system holds a local copy

of the whole application data in the main memory. The refresh aspect ensures that the local copy of the data is up to date. All the objects that reside in the main memory are unique, meaning that the application data stores exactly one copy of every object at any time. So if a remove operation is to be performed for an object in the main memory, persistent data that belong to that object in the database should also be removed. Deletion operations on the database could, therefore, trap the remove operations to be performed on the object instances and then remove the corresponding persistent data.

However, deletion by aspects does not have to be like this always, because the case specific situations mentioned above would not always hold. In [1], deletion is mentioned to be a concern that the developer should be aware of during the development of the application. This is because, the deletion should be non-fuzzy such that removal of the persistent data should be specifically declared by the application. Otherwise, deletion of an object instance may or may not require or intend deletion of persistent data. Also, automatic garbage collection mechanism of Java Language makes it hard to follow removal of object instances.

In our car rental system, the only objects that can be removed are rent and reservation, and corresponding return and payment objects. So, deletion operations on the database are applied only to the persistent data corresponding to these objects. Removal of the persistent data corresponding to shared objects should not be performed. In our case, the car rental system does not trigger any shared object deletions, however, in any other system this may not be the case. A system that has more than ten thousand lines of code could be too large to delve in to identify shared object removal triggers. This is a case in point for the requirement of non-fuzzy deletions stated in [1].

2) Synchronization

Consistency of the database is critically important for all applications which are working on database. As we know so far, the multi user systems which are working on the same data always brings the synchronization problem.

Synchronization is a cross cutting concern because it is scattered over the whole system. There can be lot of objects which has to be synchronized. For each object the synchronization concerns should implemented.

To implement the synchronization concern we used semaphores. Our semaphore mechanism works by setting a field on the database tables of each object named `locked`. When another operation tries to set the lock of the object, systems doesn't let the user to that because that object is still being modified.

Main problem is our case can be used by different users on the different branches. This means they are going to work on the same data. For example when two users log in to the system to change a car description, they should not be able to modify same car description at the same time.

For implementing the synchronization as a concern, we have to find the pointcuts first. We made the aspect mining operation by our self without using any tools because the synchronization concern is not very easy to localize. We take each modify can be done by GUI is an operation which should be synchronized. The pointcut which is defined in the first line of Fig. 8 is showing the local manager modify operation. While a user is modifying a local manager the advice shown in the fifth line of Fig. 8 is checking if that local manager is being modified by someone else. If so it gives error. Otherwise it is locking this local manager for updates. The after advice declared in the twenty fourth line of the Fig. 8 is the advice which unlocks the local manager after an update.

```
1  pointcut syncLocalManager(BranchManagerWidget caller)
2  : execution( * BranchManagerWidget.localManagerButtonClicked() )
3  && this(caller);
4
5  void around (BranchManagerWidget caller) : syncLocalManager (caller){
6      BranchManagerWidget branchManager=(BranchManagerWidget) caller;
7      int baseIndex = ( branchManager.currentPage - 1 ) * branchManager.MAX_ITEM_COUNT;
8      localManagerIndex=branchManager.managerWidgetUi.tableWidget.currentRow();
9      Branch branch = branchManager.getBranches().get( baseIndex
10     + branchManager.managerWidgetUi.tableWidget.currentRow() );
11     LocalManager localManager=branch.getLocalManager();
12
13     String updateQuery="UPDATE salesPerson SET locked=\"1\" WHERE nationalId=\""
14     +localManager.getNationalId()+"\" AND 0=locked";
15     if ( updateExecute(updateQuery)>0 ) {
16         proceed(caller);
17     }
18     else {
19         QMessageBox.critical( null, "Local manager Modify Error",
20         "Local manager is being modifiying by another user please try again later.");
21     }
22 }
23
24 after (ManagerWidget caller) returning : syncLocalManager(caller){
25     String updateQuery=null;
26
27     BranchManagerWidget branchManager=(BranchManagerWidget) caller;
28     int baseIndex = ( branchManager.currentPage - 1 ) * branchManager.MAX_ITEM_COUNT;
29
30     Branch branch = branchManager.getBranches().get( baseIndex + localManagerIndex );
31     LocalManager localManager=branch.getLocalManager();
32
33     updateQuery="UPDATE salesPerson SET locked=\"0\" WHERE nationalId=\""
34     +localManager.getNationalId()+"\" AND 1=locked";
35
36     updateExecute(updateQuery);
37 }
```

Fig. 8 Synchronization Aspect

As described above it is not an easy task to find the points which have to be synchronized in a legacy code. If it is clear that AOSD is a clear solution for this problem. The aspect code is modular and easy to maintain now but aspect code is not reusable.

3) Refresh

Main purpose of the refresh aspect is to update the information on the object by using database. As Car Rental System worked on just in a single branch, data transfer between the different users can not be established. However, with implementation of a database we can make the application work on a network of different users at the same time. We can use database to realize this network by using a refresher aspect which updates the data on main memory with the new values at the database.

This concern needs access to the all the get operations of the objects. To implement this using OOSD we need to change the existing code by adding refresh queries in get operations. As in simple manner it needs access to get methods, aspect-oriented approach is a better choice.

Current aspect gets each call for get methods of an object and applies an update to that object. If the object holds a reference to another object then we just set the id of the object to the new value from database as it can be seen in line 32 of Fig 9. When get methods for the referred objects are called their attributes will be update which will lead to a stably refreshed application.

```
1  pointcut allMethods( Object obj ) :
2      target( obj ) && ( execution( * core.*.get*(..) )
3      || execution( * core.*.find*(..) ) || execution( * core.*.search*(..) ) )
4      && !cflow( execution( public static void Fill.main(..) ) )
5      && !cflow( execution( * DataUtil.*(..) ) )
6      && !cflow( this( Persistence ) ) && !cflow( this( Refresh ) )
7      && !cflow( execution( * core.*.update*(..) ) )
8      && !cflow( execution( * *.modify*(..) ) );
9
10 before( Object obj ) : allMethods( obj ) {
11     if ( obj instanceof Branch ) {
12         ...
13     }
14     else if ( obj instanceof BranchInformation ) {
15     ...
16 }
17     else if ( obj instanceof Car ) {
18         ...
19 }
20     ...
21     else if ( obj instanceof Salesperson ) {
22             Salesperson salesperson = (Salesperson) obj;
23             selectQuery = "SELECT branch, name, surname, password, type" +
24                           " FROM salesperson" +
25                           " WHERE nationalId = '" +salesperson.getNationalId() + "'";
26
27             try {
28             query.executeQuery ( selectQuery );
29
30             ResultSet rs = query.getResultSet ();
31             if ( rs.next() ) {
32                 salesperson.getBranch().setId( rs.getInt( 1 ) );
33                 salesperson.setName( rs.getString( 2 ) );
34                 salesperson.setSurname( rs.getString( 3 ) );
35                 salesperson.setPassword( rs.getString( 4 ) );
36                 }
37             else {
38                 System.out.println( "Salesperson select query error for " +"salesperson with national
39                 +salesperson.getNationalId() );
40             }
41             }
42             catch( SQLException e ) {
43                 System.out.println( e.toString() );
44             }
45         }
46     ...
47 }
```

Figure 9: Example code from refresh aspect

As in other concerns, aspect-oriented approach helped us to modularize the system and implement the system without any change in the legacy code.

The main problem we face during the implementation of the refresh aspect is the time duration to check the database for updates. If the duration is minimal then there is a performance issue as we need to update the data for each object every time a refresh is done. However, if the time interval for refreshes is large, then we can face with synchronization issues. Since, the user can modify a data which has been modified before but has not been updated. This issue can be implemented as a synchronization problem but we did not have time to overcome that problem.

The second problem is that with the system of refreshing the ids of the referred objects we ignore the part where there are arrays of objects. If there are changes in the arrays we can not refresh those parts because we work on a single object refresh. However, for domain specific application deletion of a object or addition of a new object is not an issue. For example, adding a new car to a branch will add a row to car table of the database. On the other hand, our refresh aspect just gets the cars of the branch which were initialized at the beginning of the application but ignores the added cars after initialization. This problem can be implemented in the same way initialization aspect works, but we should check whether each object in the arrays are present in database, if not remove those objects. After this operation, check whether each row retrieved from database present in array of object, if not exist add it to the array.

4) Failure Management

This aspect mainly focuses on the database failures. It needs to access to points where database queries are called. As we implemented other database aspects this concern focuses on the aspects and tries to control the database validity before any query is executed.

To modularize the failure management concern in a specific module, other than implementing them inside other aspects such as persistence and synchronization, we choose aspect-oriented approach for our implementation.

Implementation gets the specific points where queries are executed. For each button click we check whether we have a valid connection to the database as it can be seen in line 15 and 28 of Fig 10. If there is not a valid connection, we ask for a re-connection as it can be seen in line 20 and 35 of Fig 10.

```
pointcut buttonClicked( QWidget parent ) : target( parent )
    && ( execution( * gui.*.*Clicked(..) )
    || execution( * gui.*.modifyInformation(..) )
    || execution( * gui.*.searchCustomer(..) )
    || execution( * gui.*.modifyCustomer(..) ) );

private void showConnectionFailureDialog( QWidget parent ) {
    QMessageBox.warning( parent , "Connection problem!", "Connection can"
    +" not be established! Re-try later or consult to your " +"database provider!" );
}

void around( QWidget parent ) : buttonClicked( parent ) {
    Connection con = Persistence.con;
    if ( con == null ) {
        if ( QMessageBox.question( parent, "Database Connection error!",
        "Database connection is lost! Do you want to re-connect?",QMessageBox.StandardButton
        , QMessageBox.StandardButton.No )== QMessageBox.StandardButton.Yes.value() ) {
            if ( Persistence.connect() )
                proceed( parent );
            else
                showConnectionFailureDialog( parent );
        }
    }
    else {
        try {
            if ( !con.isValid( 1 ) ) {
                if ( QMessageBox.question( parent,
                    "Database Connection is invalid!",
                    "Database connection is invalid! Do you want" +" to re-connect?"
                    , QMessageBox.StandardButton.Yes,QMessageBox.StandardButton.No )
                    == QMessageBox.StandardButton.Yes.value() ) {
                    if ( Persistence.connect() )
                        proceed( parent );
                    else
                        showConnectionFailureDialog( parent );
                }
            }
            else {
                proceed( parent );
            }
        }
        catch ( SQLException e ) {
            showConnectionFailureDialog( parent );
        }
    }
    return;
}
```

Figure 10: Example code from failure management aspect

The main problem of the current implementation is that we ignore the failure between the button click and query execution. If the system requires for a query to be executed, some time after button click we can not be sure that the connection is still valid. Therefore, this type of solution is not enough for bigger applications working on unstable database systems.

The second problem of the current applied solution occurs if there is more than one query to be executed in the database. After the button is clicked, if there is a database failure after the first query then we are expected to undone the first query. To do this, we should be aware of which type of query is executed (insert, update, delete). If it is an addition, we can delete the added object which is hold in the persistence aspect. For delete operation same mentality goes. However, for update operations we must have knowledge of old object to set back the attributes to previous values. To do that we should get the update method of the object and use a before advice, but before to do the update we get the update method and use an after advice. This means that the query always will be executed after the update method is finished, then we need to take both before and after advices for the method and keep the old object in the aspect and then use it when there is exception caused by database failure. However, this will lead to keep the all updated data in the aspect for any failure, which would cause significant performance issues if the system consist of millions lines of codes.

4. DISCUSSION

We analyzed an existing system for finding the database aspects and the implementation of the system is done by using AspectJ. In the first times, we think that it will be easy to localize the aspect from the code because our legacy code was well written and we were having all of the documentation of it. After some workings on the code, we saw that it is not easy to find the join points of the legacy code for some of the database concerns.

In our case we were using a car rental system. It was not a very complicated system. Also the legacy code was relatively short. Our technique can be considered as aspect mining on legacy code. Other mining techniques [2], [3] would not work because we were not searching for a tangling code or scattered concerns. In our approach database aspects are added as a new concern. These new concerns could be hard coded to the existing code but this would lead to coupling and low cohesion. AOP techniques should be used for having a clean code and maintainable system.

Our approach is working on the legacy code, this makes some of the concerns hard to implement. On of these concerns is the update operation of the persistence concern. Update operations can be catch by getting the set methods calls as a pointcut. When an object updated, say object has twenty features, there will be twenty different update calls to the database instead of updating the whole object. This will lower the overall performance of the system. Another issue which is similar to this exists in refresh aspect. In this aspect we are getting the get methods as a pointcut. Therefore, for each get method call of the object a select query is executed on the database. For example when user tries to search for a car refresh aspect will execute six select statements for each car. If we had more than fifty cars that will cause 300 select queries just for a simple listing operation. For having an easy coded and reusable aspect we took only the get operations for each class as a point cut for refresh aspect. But we lost the performance.

In our case, deletion part of the persistency concern was easy to adapt since the existing application does not allow any fuzzy deletions. But this is not the general case; removal of object instances would not always intend deletion of corresponding persistent data. So, deletion of persistent data needs to be explicitly declared in the application code. This requirement prevents the deletion form being fully modularized by aspects.

5. Related Work

Rashid et al. defines persistency as an aspect [1]. Their work is intended to provide an insight into the issue of developing persistent systems without taking

persistence requirements into consideration during development and adding the persistency functionality at a later time. They try to answer the question whether persistency functionality can be added to a system at later time. The conclusion they have come to, is that developers cannot be oblivious to all of the operations that are required for persistency. Still, some of the persistency operations could be left out of the development process and can be added later. Retrieval and deletion operations need to be exposed whereas store and update operations do not need to be accounted. They have also explored re-usability of persistency aspects. The suggested persistency aspect is found to be very simple to adapt and reuse; however, for effective reuse a specification that clearly defines the behavior of the aspects would be required.

Soares et al. described adding persistency and distribution aspects to an existing web based system[4]. Functionality of the existent persistency code that is embedded into the system code is replaced by the persistency aspect. Persistency aspect address following major concerns: connection and transaction control, partial object loading and object caching for improving performance, and synchronization of object states with the corresponding database entities for ensuring consistency. They identified some disadvantages of AspectJ such as lack of re-usability and proposed some minor modifications for improvement. They stated that AspectJ's powerful constructs must be used with caution to avoid undesirable side effects. Pointcuts are stated to be highly specific to a system and lacking re-usability. For these reasons, they suggest support for aspect parameterization. Persistency and distribution aspects were found to be not completely independent due to shared exception handling and synchronization mechanisms. Some of their aspects are defined as abstract and reusable, whereas others are application specific.

6. Conclusion

In this paper, aspectising database aspects such as persistence, synchronization, failure management and refreshment of the objects from database has presented. The AOP approach is usually used for eliminating tangling code but in this paper only adding the new concerns to an existing application is being considered. Modularity and the maintainability of the system considered as an important issue and not a single line of the legacy code is changed.

We had a car rental system as a case study. In this system a user was able to rent and reservation cars. The weak points of the system were: working on files and only a single user can work on the same program. The existing system should not be changed but new concerns thought to be added.

These new concerns were only about the database issues. For implementing this AOP approach is used.

We define the database aspects which should every application support such as: persistency, synchronization, and failure management and refresh operations. Each concern declared as an aspect in the system. We first find the join points for each aspect. After defining the join points the business logic of each concern coded in the aspect. AspectJ is used for implementing AOP techniques.

It this paper we tried analyze and implement database concerns for an existing code. For a car rental system proves that database aspects can be applied to a legacy system without changing its code. Also it is now easy to maintain the new added concerns.

As a future work we are planning to add a different database aspect called transactions. In this paper failure management section acts like supporting transactional behavior but real transaction mechanisms are more complicated. Also it is really hard to say that the aspects which are presented on this paper are reusable. We are planning to declare more reusable [1] aspects especially for the persistency concern. We encountered a tangling in our code. Each SQL statement could be produced according to the type of the class and the type of the operation. An aspect such, `SqlStatementProducer` can handle this tangled code in our system.

7. Acknowledgment

Thanks to Asst Prof Dr. Bedir Tekinerdogan who is the main organizer of the “Third Turkish Aspect-Oriented Software Development Workshop”, for helping us during the development of this paper and sharing generously his comments and feedbacks with us.

Thanks to all participants who attend this workshop and share their ideas.

References

[1] Rashid, A. and R. Chitchyan (2003) *Persistence as an Aspect*. 2nd International Conference on Aspect-Oriented Software Development. ACM. Pages 120-129.W.-K. Chen, *Linear Networks and Systems* (Book style). Belmont, CA: Wadsworth, 1993, pp. 123–135.

[2] A. van Deursen, M. Marin, and L. Moonen. Aspect Mining and Refactoring. In *First Intl. Workshop on REFactoring:*

Achievements, Challenges, Effects (REFACE), 2003.

[3] S. Breu. Aspect Mining Using Event Traces. Master's thesis, U Passau, March 2004.

[4] S. Soares, E. Laureano, and P. Borba, "Implementing distribution and persistence aspects with AspectJ", OOPSLA, 2002, ACM Press, pp. 174-190.