Implementation of Persistence as an Aspect

Vishal Verma
Kurukshetra University Kurukshetra
Department of Computer Science & Application
Kurukshetra University P.G Regional Centre
Jind,

Ashok Kumar
Kurukshetra University Kurukshetra
Department of Computer Science & Application
Kurukshetra University, Kurukshetra, India

ABSTRACT
Data base and its implantation through various means, always demand the security of data which is stored in it. From view point of security from earlier days the DBA stores the data at different levels and level is available to the concerned person only. The data with the highest level of security is also termed as persistence data. Some authors also consider the term persistence as proper means of storage and retrieval of data from the storage devices. During this storage and retrieval the overlapping of code for verification of user make it implementable as cross cutting concerns. In this paper we provide the inside scene for implementing persistence as an aspect so that it can be embedded with phases of software development and can be reused when needed.

Keywords
Aspect Oriented Programming, Aspect Reuse, Persistence, Relational Data Base.

1. INTRODUCTION
The term aspect in Aspect Oriented Programming (AOP) [22] provides the basic means for designing modules for a software problem. Concerns like synchronization [18, 5] and tracing [19, 7] are used to describe the aspectisation. On the similar pattern persistence is also considered as an example [11, 10] of concerns. This supports that persistence can be modularized in AOP technique and hence can be reused. Further applications (without taking persistence into consideration) can be developed to the fact that the synchronization tracing and persistence aspect may be composed at later stages. Though we are able to claim all of the above facts but still we can't express this all by considering examples, some of the important unexplained points are:

a) Modularization of aspects can be done by using AOP.
b) Aspects used to represent persistence can be reused.
c) It is possible to develop applications without filtering data as persistence or non-persistence.

Some of the existing projects [1, 2] in AOP have taken into consideration the persistence and related concerns for designing an approach and prototypes etc to store the aspects into data base. AOP uses the persistence representation for aspects to keep this representation independent of any particular approach. Main concentration is on providing aspect persistence along with the persistence of application data. Similar kind of techniques is also discussed in [3] but separation of persistence is not taken into consideration. In [9] an approach for AOP is discussed which is based on separation of concurrency control and failure handling code in relational (distributed system). The case study in this paper is mainly concentrating on aspectisation of transaction (in banking system) which is implementing the persistence in real. We are discussing only model (schema) of transactions without any consideration of actual code for the same. All transactions under consideration here are treated as an object and support to operate in object oriented environment. This is used most commonly in Data Base application currently used in corporate sectors. In [20] there is a description of implementing persistence and distribution aspect with AspectJ. The aim is to refractor the existing application, instead of exploring any approach for application development which is independent of persistence requirements or developing any new reusable persistence aspect.

This paper shares the experiences in representing the persistence as an Aspect with the aim that is it possible to make aspect in a real world application against persistence. More importantly we try to find whether aspect presentation like this can be reused with the application and aspect which is developed independent of each other. For the purpose of case study we assume the basic transaction processing system of a bank and its model implementation in SQL to properly consider the underlying persistence mechanism. We try to provide the general insight into applicability of other AOP techniques in this context and discuss how this model can be adapted to apply on other data base technology i.e. Object Oriented Database etc. Assumed data base structure for bank:

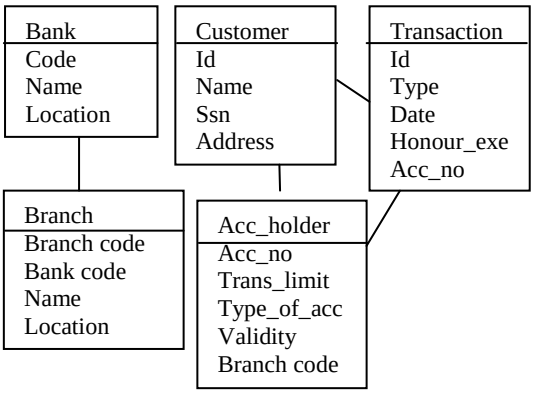

Figure 1 Bank's Data Base Structure (UML Notations)

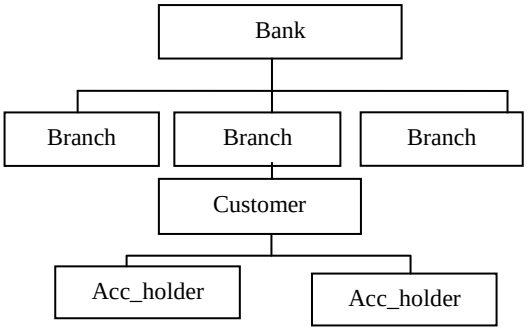

Figure2 (Contd.) Bank's Data Base Structure (UML Notatons)

Data model for the application is designed by assuming the basic transaction processing among various elements for the purpose. It is one of the simplest in nature as we don't want to consider the actual relationship among the tables. It is also better idea to consider them as view instead of tables for the purpose. Data model is represented by UML notations and is shown in Figure 1. Aspects are used to implement the associations and aggregations when appeared in the model.

2. MODULARIZING PERSISTENCE

From the beginning of this section we specify the rationale behind the design and approach used for accessing data base. During development phase of software we are not paying attention to the need of performing operations like store and update in the data base. However every operation component is accountable for both the data storage and nature of retrieval. Hence we pay less concentration to modularize the retrieval process. Accordingly deletion is assumed to be performed individually by separate module.

2.1 Data Base Access

Three important considerations at the time of aspectising the data base access required for an application which don't consider the persistence are:

a) The way it is identified to separate the persistent data from application (operational) data.

b) The reusability factor at the time of data base aspectisation is important.

c) To fully support the P2 there must be some plug-in points by using which the requirements of new application in which aspects is to be reused can be embedded easily.

To separate the persistence data from application (operational) data we use the persistence root-class from Object Oriented Database [15]. These type of system also demands the existence of a common base class for all classes whose instance are required to be stored in the data base. All of the base class's functionality which bears some persistence related data is augmented to persistence class by a pre or post compilation processor. The root class of a persistence class is shown in Figure. 2. We include the functionality of delete operation within this class simply to avoid any kind of dangling reference and it is obvious not to ignore the deletion like other operations on data base. Hence this all is simplified by embedding the functionality within the persistence root class along with data base access mechanism.

```
Public class RootClassforPersistence
{
Protected Boolean deleted: False;
Public void del ()
{
This. deleted = true;
}
Public Boolean deleted()
      {
      Return this. deleted;
      }
}
```

FIGURE 2 Root Class for Persistence

```
Public aspect APDB Access
{
Declare root: (Bank || Branch || Customer || Acc_Holder)
Extends RootClassforPersistence;
//other statements;
}
```

Figure 3 Super Class (An Aspect Declaration of RootClassforPersistence) with persistent instance

Here the RootclassforPersistence plays very important role to aspectising data base application access in a reusable fashion. In Figure 4 we show the Aspects for Data Base Access with high degree of reusability which is implemented by defining the join points with reference to an application independent point i.e. RootClassforPersistence. By using this RootClass as a base class in any other application and declaring its subclasses or by declaring aspect with respect to this base class we can implant the reusability.

```
Public abstract aspect DBAccess
{
I   Private static Conn dbconnection;
    Private static str  dbURL;

    Abstract prointcut estbconn();
    Abstract pointcut clsconn();

    Public abstract str getDBURL();
    Public abstract str getDRVName();

II  Pointcut trapInst(); call (RootClassforPersistence);
    Pointcut trapVpd (RootClassforPersistence Ob);
    flow (call (public static vector SQLTranslation getobj
    (Result.Str)))  && (this(obj)) && execution (public
    void RootClassforPersistence.set()));

III pointcut trapRet ();
    call (vector PersistentData.get(…));
    public static PersistentData  getPersistentdata {}

IV  pointcut trapDel (RootClassforPersistence obj) :
    this(obj) && execution (public void
    RootClassforPersistence.del());
    pointcut Finddelobj (RootforPersistence obj) : this(obj)
    &&
    (execution(public * RootforPersistence *.get*.(..))||
    execution(public * RootforPersitence * .get*())||
    execution(public *.RootforPersistence*. tostr()));

V   persistent static Integer upd(string sqlstate)
    throws SQLExcp {….}
    protected static vector retrieves (string sqlstaic.string
    classname)
    throws SQLExcp {…}
    protected static object transactionwrapper (string
    methodname object parm){…}

VI  public static aspect metaDataAccess {…}
    //other code
}
```

Figure 4 Features of DBAccess object

2.1.1 Conn

The connecting and disconnecting from a data base must be the basic feature of persistent application. However, we have already discussed that this type of feature is generic and is implementable only with availability of some special customizable point cuts with the class. These point cuts must be in correspondence to:

a) The geographical location of database.

b) The database management system and data base engine used for data access.

c) The available points in the whole data base generally considered as entry and exit points from where new connection with existing DB can be established and disconnected.

In the DBAccess aspect we implement/define the above requirements by two abstract point cuts and methods. Abstract methods are supposed to be called by a *before* advice which operate on abstract point cut *estbconn* to get information required to establish connection with data base (static type variables are used to hold information about connection) . There is also a need to supply the data base URL and driver detail form the application who is supposed to call the DBAccess aspect and two related methods. This type of aspects also concerned two point cuts for specification of join points in the statement blocks at which DBConnection are supposed to be established (i.e. entry point) and

closed (i.e. exit point). For the implementation of the same we opt to use the APDB Access () from Figure 3. Any kind of connection pooling is not implemented herewith; however, the same can be called into the DBAccess aspect with localized impact. The availability of driver and connection managers is quite common through the ODBC or JDBC drivers, there are some limitations on JDBC API that they do not provide the full support for application which may demand the access the to the meta data. The application which support the flexibility in choosing the data base on which it operate must provides way of selecting the different data base drivers/connection managers as well.

2.1.2 Storage and Updation

For the purpose of storage and updation we use two different point cuts which identify the locations where an object is to be stored and corresponding persistent representation/data is updated. Call to trapInst by any application shows that the corresponding object is instantiated and it should be updated in the data base immediately. At the time of implementation of aspects for storage and updation two imported factors must be given the proper concentration.

a) Proper concentration must be given to the persistence by reachable [16] which stress on the fact that the entire object which are reachable from the object to be stored must be made persistent. It is needed to ensure the fact that all references can be reestablished on retrieval. Since the data base under consideration is relational data base hence the implementation of this fact is left to the SQL translations i.e. insert statement.

b) For storing an object it is an oblivious condition that its respective constructor got executed first, this implies that only *after* advice executed. But care must be taken in case of rollback transaction, in this case the object's instantiations will not got aborted itself (automatically) and should be managed by application. Hence on the execution of transaction wrapper that signals the rollback, its corresponding *after* advice statement must ensure that appropriate action be taken for this rollback or the respective record from the data base must get deleted to avoid the dangling references in memory. It is also submitted here that advices must be treated as first class entities for proper matching of signatures of the behavior specified within an aspect.

The execution of update method relies on call of all appearance of setter methods on persistent objects. The appearance of such call within the statements of getobj method of SQL translation aspect is not considered for execution. This type of methods is used to build the objects to be used in relational representations. Hence call to setter method in the set of statements, is used to fill up the empty copy corresponding to the object and, hence, do not have persistent perspective. In this case a *before* advice, may be employed to make confirm that data base state get updated before updation of transient object. This implementation helps to ensure the error/problem free execution of roll back operation on data base. In our implementation we also avoid any public access of any of the method all are supposed to be accessed via get and set methods etc. This practice helps to ensure the one fix interface for class (not modified frequently) against changed interface of member variables. However, to make the application more prominent only trapupd point cut definite should be modified to trap direct updates.

The trapInst and trapupd point cuts do not demand any specific preparation in the code of application that instantiate the classes in Figure.1. It makes the developer free from the fact that all advices which are in need to refer these point cuts will store and update objects in data base and their respective persistent representation.

2.1.3 Retrieval

The retrieval operation is most important operation among all other operations performed on data base since the requirement of the end user get satisfied only on proper execution of retrieve operation and hence it is important to pay proper concentration to the successful execution of this operation. The term retrieval can be specified as "to get and bring back: especially: to recover from storage"[19]. Form this definition it is obvious that data is retrieved from one source and is used at other source. It is further modified that the operation retrieval of data is of declarative in nature since it may include the predicates and selection conditions (simple/complex). For all type of data base query languages are used to access the data dominantly than other methods. In case of object oriented data bases the data can be retrieved only by traversal i.e. by using Object Query Language which is a part of ODMG standards [15] or by implementation of a proprietary query language as commercially discussed in [21, 23, 13, 14].

Though all above facts are true, the aspects can play an important role for designing/developing code for retrieval operation. We implement it by a special interface termed as PersistentData which offer methods to be implemented by classes. All the retrieval of data based on any kind of conditions. All of these method return the object in form of vector. getPersistentData () method of the DBAccess aspect always provides a reference to the instances of classes which implements this interface. Application is suppose to obtain this reference execute all of the retrieval related code.

Any class which invokes and implements the PersistentData interface provides the points to be used by trapRet point cut for identification of points at which application is suppose to make entry for data retrievals. It clearly reflect that this is not application dependent and hence is reusable around advice for trapRet point cut is used to obtain the conditions which are passed as arguments to hook methods. By using SQL Translation it is supposed to search and return the data. Because of application independence above discussed methods provides the high degree of reusability. Since all the data base based applications necessarily use the retrieval method hence it automatically become important for all applications. Design of user interface component may depend on the amount of data to be retrieved from the data base. Numbers of other design issues are also considered for designing the interface. It provides means for relating a Acc_Holder to Branch and a customer to a Bank etc. This type of interface is necessary to provide since a bank may have thousands of customers and Acc_Holders who may demand various type of operations to be performed (retrieval data) and must be managed properly. Even on selecting a branch the population of Acc numbers is very high. Hence it seems very meaningful to provide the means ot the user by using which the name of branch or customer or Acc_Holder be selected by using graphical interface and the retrieved data is of total interest. This all is considered as data optimization and the results provided by this are manageable in an easy way.

2.1.4 Delete

The delete operation is also implemented for all data base applications and as discussed above it is not possible to fully aspectise it among the applications. The aspectisation of delete operation is not possible since it is required to be executed on request from the specific users only and hence there are implementation dependent factors which play important role for it. Some of the languages provide the automatic handling of garbage collection and dangling reference hence their programmer need not to pay special concentration for implementation of this operation. Hence it is not needed to provide any reference point DBAccess aspect to remove a persistent object from data base file. In languages like C++ the call to delete operator or aspect may be implemented to remove object from data bases. But mere call to such operator provides no means to ensure whether the object is to be deleted from the memory or from the data base itself. This all support the implementation in which it is better to call the delete as an explicit method (reference point) at which any other aspect can

operate when required. In the base class discussed in Figure. 2 we provide such type of reference point of delete method in the RootClassforPersistence. Any of the application which want to execute this method can call it by declaring a new sub class for this super calss and invoking the method within the boundary as shown in Figure. 3 the trapdel() point cut discussed in Figure.4 is supposed to capture any invocations along with *before advice* and remove the persistent representation of object as is done in case of update operation. It is designed to do all necessary implementation for removing all the early collection of objects to avoid any kind of dangling reference or to properly implement the garbage collection. Measures are also taken to properly through and catch the exceptions (if any occur). This declaration of delete () method in the RootclassforPersistece (as a reference point) make it implementable of this code as a reusable code when needed. But for it to be reusable, the programmer of the application has to pay proper attentation toward the existence of base class which should contain the same declarations for it as in RootclassforPersistence, and then define the subclass which implement this method as per the request of the application and data base. The existence of base and subclass combination is necessary to avoid any kind of ambiguity in the application and by doing so the programmer can also make himself/herself free from the functionality of DBAccess aspect and any other SQL Translation aspects.

2.1.5 Transaction

The term transaction in itself means a lot for data base application since it includes the methods like addition, updation , deletion, retrieval and transactionwrapper etc. it is better to include the transaction functionality within such space if possible. The single spaced wrapped up method's availability is also supported by easiness like JDBC in which operations/transactions are started implicitly (always). The transactions like update and retrieve are corresponding to read-write and read any operations in the data base respectively. It is clearly reflecting that update, addition and deletion transactions always cause some changes in the present state of data base and are implemented through respective SQL's queries by the user. The respective SQL queries are used as an argument to the point cuts where we want to implement the above transactions. Name of class is used as an argument to all methods in classforPersistenceData interface and it is used to establish relationship/mapping between structure (object) and relational structure i.e. data base.

```
Protected static object  transactionwrap (string methname, object[] parameter)     {
Try {
Boolean Iscommit = true;
Object obj = Null; }
                Try {
                Class this class = class for name ("DBAccess") ;
Method [] methods = thisclass.getdelemeth();
Method themethod = Null;
For (int I = 0; i<methods.len; i++) {
If (methods[i].getname()==methodname)
theMethod = Methods[i]; }
Obj = theMethod.call(Null,Parameter);
}
      Catch(Exception e)  {
      System.out.println (e.tostring());
      Dbconn.rollback();
      Iscommit = false; {
      Finally {
      If (iscommit)
      Dbconn.commit();
      Return(obj);     } }
      Catch (SQLException e) {
      System.out.println ("Error in commiting for rolling back" converttostring());
      Return Null; } } }
```

Figure 5 Method showing Wrapping of Transaction

Update and retrieve methods may not accessed directly by all devices, rather it is implemented as name of a number methods which are passes as an array to transaction wrapper method. This all helps in easy modularization of nested try catch blocks, otherwise they are required to be implemented as individual modules. Outer try catch block take care of and handle the SQL exceptions when commit and rollback methods were called. Inner block of try catch invokes the required methods. This all is similar to the method discussed in [9] where it was used with a Boolean variable to decide whether to commit or roll back. Abort of transaction alternate is opted if transaction is reflective transaction or data base operation. This safer option is chosen as any reflective operation plays fundamental role at the time of translation to/from SQL. Chances of relating an exception with data base (directly or indirectly) are higher. Almost all the functions related to access of data base are aspectised. Hence there is no need to raise exception at the time of abort operation since same signaled and well managed by aspectisation infrastructure. At any stage the returns of value NULL implies the unsuccessful transaction, and give signal for execution of transient rollback. The method used in [9] for transaction wrapping is not used in this paper rather we call the transaction wrapper (explicitly) from the advice code that were designed for storage, update, retrieval of simple and persistent objects. It make the strict boundaries for calling and execution of transaction wrapper method against data base operation only and not for other transient operation. Hence the aspect design is supported by the fact that it is not operating n pure Object Oriented environment. Reflection operation of data base demands some overhead for its proper implementation. A good combination of read – write and read only locks is provided for update and retrieve methods optimization.

2.1.6 Meta Data Access

This aspect is static in nature and it encapsulates the helper functionality to access the meta data i.e. schema related data like table name, column name etc from data base. This type of data is actually required by SQL transaction aspects. It is developed to fulfill the two purposes as

a) To avoid any kind of duplication of data.
b) If any Meta data method is not get support for related data base drivers then the same can be built easily on top of more primitive features. Hence all update operation can be carried out without affective the SQL translations function when new version of drivers are available.

It is quite clear that meta data access functionality can be treated as a subset of all functionality of data base access. The module corresponding to this function as inner aspect of DBAccess aspect provides more natural creation of concern.

3. SQL TRANSACTION

Transaction of SQL statements considered as separate concerns at the time of aspectising of persistence in relational data base; this is because all data base access is treated as a concern for all applications involving access to persistent data. However, any translation of underlying data model is not necessary. Whenever relational data base persistent is accessed by OO application the object structure of relational data base is flattered for easy access since it may not be able to provide the full support for complex data types. In Figure. 6 we are showing a part of our relational data base.

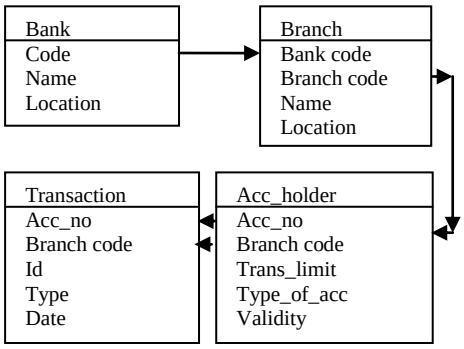

Figure 6 Mapping of Object Oriented Data Model to Relational Data Model

Here the object Bank and its branch are taken as two separate tables and are co-related by Bank code so that all branches can be identified separately and remain related to the Bank all the time. The tables as shown has one to one relationship among Bank, Brach and Acc_Holder. Further one to many relationship can be maintained easily among the tables Acc_Holder and Transaction and are implemented by using Branch Code and Acc_No in both tables together. The important part of implementation demands the existence of a mechanism to create the related tables from the object data base. Some time the use of only JDBC drivers is augmented and is stressed for use but it may raise the complex situations when two way movements for the search of data is not supported by this kind of drivers and there use has rather limitation of providing results as a Result Set and all updates demands the use of unnecessary disk spaces. The SQL Data Interface in JDBC is able to supports mapping to/from only for user defined object relational data model and not for pure relational data bases.

For any SQL translation to be reusable the minimum condition to be satisfied is that it must not be dependent in any application. As discussed before any kind of mapping required for application must be specified at the time of aspectisation of persistence for application. For the purpose of mapping we describe a singleton lookup table. Mapping is maintained only for many to many maps. The one to one mapping is simple to maintain as one can be done by using the identical names on both side. But it is compulsory to maintain the different name scheme that we have to follow the lookup table for one to one as well as many to many cases. To implement the mapping in the look up table we use Create map aspect which setup the mapping before establishing the connection with the DB .

```
Public aspect createmap dominates DBAccess {
Point cut setmap();
Application DBAccess.establishconn ();
Before (); setmap () {
Lookup table maptable = LookupTable.getLookupTable();
Mapping table.create classto Table Map (“Bank”, “Bank”);
Mapping table.create classto Table Map (“Branch”, “Branch”);
} }
```

Figure 7 Object-to- Relational Mapping (By Aspect)

As Bank object is mapped to Bank Table, Branch is mapped to Branch Table and so on. It is important to note here that create map aspect must denote the DBAccess aspect just to ensure the establishment of mapping before any kind of connection with DB. Main features of SQL Translation are shown in Figure. 8. We create a sqlExe point cut to handle the fact that an object of data base may map to many tables and hence result in translation to more than one SQL statements. An around advice to test whether a single SQL statement is executed if so then normal execution of DBAccess aspect is supposed to proceed. For execution of more than one statement it’s better to use execution in batch processing. For execution of all statements of SQL it is better to treat them as concern. The sqlExe point cut captures Stetement.exeupd(String) calls for a single method in the DBAccess it becomes possible her to separate out the SQL translation functionality and add it up with the SQL Transql aspect. Almost all the get/SQL and get/Object methods finds the mapping information from the lookup table for mapping of objects and respective update, retrievals from the data base and if needed also handle their recreation. Because of total encapsulation proper care must be taken to include all the declared methods and members. If there is provision for propagation of updates for all table linked with each other then this feature is exploited otherwise tables are updated individually. This all must be implemented within single boundary. Use of reflection for object to relation table mapping may cause some additional overheads during data base interactions, but as per [6] such issues are considered only when designing highly flexible components.

```
Public aspect SQLTrans {
    Pointcut sqlExe (statement stm, string sqlstm): target (stm)
    && invoke (public int statement.exeupd(string)&&
    args(sqlstm));
    Public static string getInsertSQL(for RootClassforPersitence
    obj);
    Public static string getupdSQL (RootClassforPersistence obj,
    string methodname , object arg);
    Public static string getDelSQL (RootClassforPersistence obj);
Public static string getQuerySQL (string classname, string
selectcondition);
Public static vector getobj (Resultset rs, string classname);
// other methods
            }
```

Figure 8 Feature for SQLTranslation Aspect.

4. EMERGING PERSISTENCE FRAMEWORK

Discussion completed in sec 2.1 and 2.2 shows that the merging persistence framework based on aspects. Using the UML notations frame work is being shown in Figure. 9. Here we are omitting the related members for the sake of simplicity. The framework shown in Figure 9 gives a challenge to some of non understandable challenges of AOP, that a module is a piece of large number of coding statements, but as per Figure. 9 this is not true for the simplest cases. It shows that for aspectisation there is need of coherent set of modules (of classes and aspect) together along with crosscutting concerns. It leads to the conclusion that aspectisation is a natural phenomena and can be achieved by separation of concern e.g. Separation of DBAccess and SQLTranslation aspects. Furthermore it’s also becomes possible to draw upon established best procedure and guidelines.

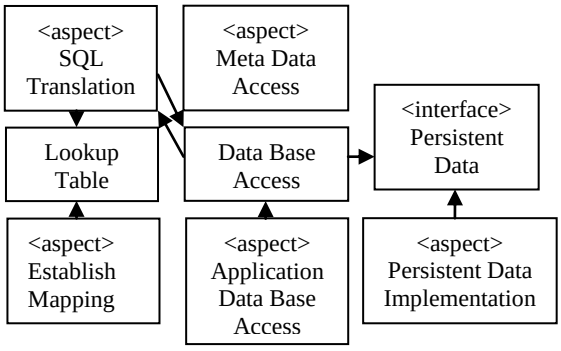

Figure 9 Persistent Framework Emerging From the application

Other Persistence Mechanism frame work discussed in Figure 9 are developed by using the assumed application of data which is in use from long time span. The basic structure with little modification can be reused in many other applications. Mainly the SQL Translation mechanism is required to be modified as per the need of target application. But in context of object oriented data bases almost everything is required to be re-implemented for new application. However, it is possible to exploit the persistence model. Almost all of the point-cuts available in the framework are used since they provide the actual entry point for the application regardless of persistence nature of target application. Similarly wrapper for wrapping transaction will be designed along with RootClassforPersistence. During this while implementation of reusability the elements like SQLTranlation. Establishmapping and lookup table are not re-implemented as we don't want to handle/create any mismatch among object oriented and relational data base. The Meta data is also not considered as there is no need for same in SQLTranslation. The RootClassforPersistence is not required to be modified as some of the propriety restrictions are applied on it to keep it as a very base class on which inheritance is based for the application. This approach works well and some was used in [25, 26] in past too.

5. RELATED WORKS

Generic persistence object has been discussed and implemented in [17], it is basically a framework for dynamic AOP. As per [20] the basic structure used for this aspectisation uses the existence of some additional layers between persistence storage and JAC. In [8] there is a description about simple data base application in which aspects are used for authentication, catching, for exception handling for pooling etc. all the storage and retrieval related statements are not aspectised but hard coded.
The discussion in [4] also bears some resemblance with the work discussed in our paper. Out discussion has persistence related domain in it which were not discussed in [12].

6. CONCLUSIONS

We presented in this paper the aspectisation of persistence in context of banking management system. Main aim of this work is to explore the support by AOP's aspects for persistence modularization and the whole discussion reply "yes" on our approach. However, in some part of this aspectisation the implementation of software engineering trade off may not be possible. The work in this paper is also aimed to implement the reusability for persistence aspects. The framework for persistence emerging from discussed work demonstrates that is indeed the case. There is no existence of any layer which is used for masking the relational data features. This framework is quite simple for adaption and reusability specifying the EstbMap aspect and use the persistent Data interface for a number of purposes. For this it must be complemented with some specification. This type of specification should define clearly the interface of an aspect's behavior.

7. REFERENCES

[1] A.Rashid, 2000 "On to Aspect Persistence", GCSE Symp.,. Springer Verlag, LNCS 2177, pp 26-36.

[2] A.Rashid, 2002 "Weaving Aspects in a Persistent Environment", ACM SIGPLAN Notices, (Feb. 2002).

[3] A.Rashid and N. Loughran, 2002 "Relational Database Support forAspect-Oriented Programming", Proceedings of NetObjectDays,.

[4] C.Constantinides, A.Bader, T. Elrad, M. Fayad, and P. Netinant, 2000 "Designing an Aspect-Oriented Framework in an Object-Oriented Environment', ACM Computing Surveys, 32(1),.

[5] D. Holmes, J. Nobel and J. Potter, 1998 "Towards Reusable Synchronization for Object-Oriented Languages". ECOOP Workshop on Aspect Oriented Programming..

[6] D.Parson, A.Rashid, A.Speck and A.Telea, 1999, "A Framework for Object Oriented Frameworks Design", TOOLS Europe, IEEE CS Press, pp 141-151.

[7] G.Kiczales, E. Hilsdale, J. Hugunin, M. A. Kersten, J. Plam and W.G. Griswold, 2001 "An Overview of AspectJ", ECOOP, , Springer Verlag, LNCA 2072, pp. 327-353.

[8] I.Kiselev, 2002Aspect-Oriented Programming with AspectJ: SAMS,.

[9] J.Kienzle and R. Gurerraoui, 2002 "AOP: Does It Make Sense? The Case of Concurrency and Failures", ECOOP, , Springer-Verlag, LNCA 2374, p 34-61.

[10] J.Suzuki and Y. Yamamoto, 1999 "Extending UML for Modelling Reflective Software Components", International Conference on the Unified Modeling Language (UML),.

[11] K.Mens, C. Lopes, B. Tekinerdogan and G. Kiezales, 1997 "Aspcet Oriented Programming Workshop Report", ECOOP Workshop Reader, , Springer-Verlag, LNCS 1357.

[12] Merriam-Webster, 2002 "Merriam-Webster Online Dictionary", http://www.m-w.com/,.

[13] Object Store C++ Release 4.02 Documentation: Object Design Inc. 1996.

[14] POET 5.0 Documentation Set: POET Software, 1997.

[15] R.G.G Cattell, D. Barry, M.Berler, J. Eastman, D. Jordan, C. Russel, O. Schadow, T. Stenienda and F. Velez, 2002 The Object Data Standard: ODMG 3.0 ; Morgan Kaufmann,.

[16] R. Elmasri and S. B. Navathe, 2000. Fundamentals of Database System (3rd ed); Addison-Wesley,

[17] R. Pawlak, L.Seinturier, L. Duchien , and G. Florin, 2001 "JAC: A Flexible Solution for Aspect-Oriented in Java", Reflection Conf., , Springer – Verlag, LNCA 2192,pp 1-24.

[18] S.Clarke, 2000 "Designing Reusable Patterns of Cross-Cutting Behaviour with Composition Patterns". OOPSLA Workshop on Advanced Separation of Concerns,.

[19] S. Clarke and R.J. Walker 2001 "Composition Patterns: An Approach to Designing Reusable Aspects", ICSE,.

[20] S. Soares, E.Laureano, and P.Borba,, 2002 "Implementing distribution and persistence aspects with AspectJ", OOPSLA, , ACM Press, pp174-190.

[21] The Jasmine Documentation, 1996-1998 ed: Computer Associates International, Inc & Fujitsu Limited, 1996.

[22] T. Elrad, R. Filman and A. Bader (eds), 2001 "Theme Section on Aspect –Oriented Programming".CACM, 44(10),.

[23] The O2 System - Release 4.02 Documentation: Ardent Software, 1998.