Towards Automatic Integration of the Business-Data Layers in Enterprise-Systems

Mira Balaban\(^1\) and Lior Limonad\(^2\)

\(^1\)Computer Science Department (mira@cs.bgu.ac.il), Ben-Gurion University of the Negev, Beer-Sheva 84105, ISRAEL. (972-8-6472222)

\(^2\)Sauder School of Business (lior.limonad@sauder.ubc.ca), The University of British Columbia, Vancouver, BC V6T 1Z2, CANADA. (1-778-558-4949)

Abstract

Enterprise information systems distinguish the Domain layer that handles the major business logic of an application, from the Data (Persistent) layer that handles storage concerns alone. The integration of these layers is not straightforward since usually the requirement is for partial persistency, i.e., persistency is required only for a subset of the Domain layer classes. Industry tools provide partial help by supporting convenient abstractions on top of concrete database systems. Nevertheless, the developer still has to design the concrete ties between the layers. Full automation can be a great improvement in complex system development, and goes along with the Model Driven Engineering approach.

In this paper we introduce a set of independent Data Access Patterns that provide the missing link towards full automation of the Domain-Data layer interaction. Each pattern is a simple refactoring that yields some Data Access layer constructs, and relies on local information in the Domain layer. All patterns are Domain layer transparent, i.e., their application leaves the Domain layer intact. We provide an algorithm for combined pattern application and show its correctness. An implementation of our method is on the way.

Keywords: Persistency, Design Patterns, Transformation Framework, Model-Driven Approach, UML, Models Co-evolution, Data Source Layer, Domain Layer, Composite Transformations, Refactoring.

1 Introduction

Enterprise information systems distinguish the Domain layer that handles the major business logic of an application, from the Data (Persistent) layer that handles storage concerns alone. The integration of these layers is not straightforward since usually the requirement is for partial persistency, i.e., persistency is requested only for a subset of the Domain layer classes. Therefore, while objects of persistent classes are stored in a durable storage, they still need to interact with objects of in-memory classes, and provide their regular services. Moreover, the integration is expected to be transparent, i.e., leave the two layers intact, and do not change the original behavior.

Domain-Data layer integration entails duplication of persistent elements between the layers. Figure 1 illustrates the complexities caused by partial persistency:

1. \textit{Inter/intra-layer consistencies} (e.g., the inter-relationships between the stored \textit{Element1} and \textit{Element3} to their in-memory copies, and the intra-relationship between the two copies of \textit{Element3} in the Domain layer).
2. \textit{Mapping} the linkage of duplicated elements on a semantic basis.
3. In-memory availability of stored elements.
4. \textit{Mixture} handling of in-memory storage interactions.

\(*\) Partially supported by the Lynn and William Frankel center for Computer Sciences, and by the Paul Ivanir Center for Robotics and Production Management at Ben-Gurion University of the Negev.
5. Behavior preservation.

Existing technologies offer partial solutions for persistency insertion. Tools like Java's Hibernate and Firestorm assist the developer in creating concrete mappings between elements of the Domain and the Data layers and provide convenient abstractions on top of concrete database systems. Standards like Sun's JDBC, JDO, DAO and J2EE, and their corresponding industrial implementations, support duplication handling and connection of an application to a storage system.

Nevertheless, in spite of the wide technological support, Domain-Data layer integration cannot be automated because the in-memory — storage mixture still requires manual provision of concrete ties between the layers, taking care of interactions between persistent to in-memory objects, accounting for collections of persistent objects, and handling of inconsistent duplication problems. These ties are the essence of the partial persistency problem. In order to solve the problem, a developer has to scan all points of in-memory – data interaction and provide concrete ad-hoc solutions for each case. This activity involves writing technical, annoying, potentially bug-infected code, and every developer would gladly avoid it.

This paper describes an algorithmic solution to the partial persistency problem. The solution consists of a small set of Data Access Layer (DAL) patterns, and their integration in a DAL-insertion algorithm. The patterns are condition action rules that solve local problems of in-memory – persistency interaction. Their combination in the DAL-insertion algorithm provides an overall solution to the partial persistency problem.

The suggested solution has the following distinguished characteristics:
- The solution is completely algorithmic (i.e. can be automated).
- Input is local. It includes marking of persistent classes and of collections of persistent classes. No global semantic analysis is required.
- The integration is transparent, i.e., leaves the domain layer intact. All new constructs are part of a new Data Access Layer.
- The suggested pattern set is minimal, i.e., achieving partial persistency might require application of all patterns.

The solution is verified with respect to three basic requirements: (1) partial persistency, (2) transparent inter-layer integration, (3) behavior preservation. We claim that the Dal-Insertion algorithm provides the single missing link towards full automation of the Domain-Data layer integration. The automatic integration takes the following steps:

1. Domain layer marking of persistent classes and of collections of persistent classes.
2. Provide Data layer information:
   a. Mapping of domain-layer classes to storage elements (e.g., Tables).
   b. Access specifications (e.g., Driver).
3. Application of the DAL-insertion algorithm.

Mappings and problems of object-relational mismatch have been widely discussed in the literature [1, 2, 5, 6, 7, 10, 11, 12, 13].
The result of the automation step is a new Data Access Layer. Full automation goes along with the rising Model-Driven Engineering (MDE) approach [9] that strives towards automatic development of software from models. Our work can be viewed as a continuation and further development of Fowler's work [3], that suggests two patterns for the integration of a persistent layer with a domain layer: The active record and the data-mapper patterns.

Due to space limitations, patterns are very shortly described, emphasizing mainly the problems that they solve and the essence of the suggested solution. A detailed description appears in [8]. Patterns involving inheritance structures are not described. Section 2 introduces the core Proxy Data-Mapper pattern, that handles a single persistent class in a context independent way. Section 3 shortly describes the rest of the DAL context dependent patterns. Section 4 introduces the DAL-insertion algorithm, and argues for its correctness. Section 5 is the conclusion, and includes a discussion of related problems.

2 Proxy Data Mapper

Consider an email client application, as illustrated in Figure 2. The example includes classes that are marked as persistent, and demonstrates inter-relationships between in-memory clients to persistent classes, outgoing references from persistent classes to in-memory classes, and relationships between persistent classes. The Proxy Data Mapper (PDM) pattern handles a single persistent class, in a context independent way. The interactions of persistent classes with other classes are handled by the context patterns. Therefore, we concentrate here on the Message class alone.

The PDM pattern can be viewed as an elaboration of Fowler's Data Mapper pattern, in a way that leaves the Domain layer classes intact. Following the GoF spirit [4], it combines the notions of the Proxy pattern and Fowler's DATA MAPPER pattern. The solution, called Proxy Data Mapper (PDM), is presented in Figure 3. It includes the following components:

1. An extracted interface (Message): An abstract class that contains all public method signatures extracted from the original persistent class with the addition of factory methods.
2. An implementation class (MessageImp): The original persistent class, with a modified name (postfix 'Imp').
3. A proxy class (MessageProxy): Represents lightweight instances of the persistent class. Each instance of the proxy-class contains an ID attribute for identifying the persistent instance in-storage. The Proxy class provides a set of delegation methods.
that correspond to public methods of the persistent class.

4. A datamapper class (MessageDataMapper): As in fowler's DATA MAPPER, this class is responsible for wrapping all database services (SELECT, INSERT, UPDATE and DELETE), and for construction of full-weight, in-memory representatives of the persistent class (MessageImp) instances.

Figure 3 – Structure introduced by the Proxy Data Mapper pattern: Combining the Data Mapper and the Proxy concepts

The main idea in this solution is replacing an in-memory fullweight instance of the persistent class with a thin in-memory representative (instance of the proxy class) enabling its full existence within storage. The in-memory representative existence is hidden from the client, being a subtype of the original persistent class interface. For example, in the email client application, clients of the persistent class Message (e.g., VirusScan) communicate with the concrete message (MessageImp in the figure) through the interface. Therefore clients are not affected. The "trick" here is that the real object that clients communicate with is an instance of MessageProxy, while their type signature knows only about Message.

At runtime, when a persistent class instance is required to activate one of its services, its proxy representative notifies the datamapper class object to re-construct a full-weight in-memory duplication of the persistent instance (a MessageImp object), and invokes the requested method implementation on it. When the service activity terminates, the proxy instance saves the persistent instance state back to storage (through the datamapper) and releases the duplication from memory. The new MessageProxy class is responsible for the synchronization of in-memory duplications and their persistent object origins. Different synchronization strategies (e.g., on-change or on-idle) may be implemented by the Proxy class. In-memory duplications do not occur because the datamapper object uses an identity map to ensure that each persistent object is loaded to memory at most once. The hiding of full persistent objects behind their proxies is actually carried out by factory methods that must replace all constructors of the persistent class, and returns proxy objects (e.g., MessageProxy).

The PDM pattern wraps classes that are marked as persistent with persistency services. The comprehensive solution to partial persistency requirements is achieved by combining it with the Context Data Access Patterns that solve the context requirements of ingoing and outgoing references. These patterns are shortly described in the next section.
3 DAL Patterns

The context of a class that is marked as persistent is determined by a combination of three parameters that gives rise to five context data access patterns, listed in Table 1:

1. **Navigation**: The direction of a reference to or from a context class (incoming or outgoing references).
2. **Cardinality**: The cardinality of the reference, i.e., does it involve a single referenced object (cardinality 1) or a collection (a many cardinality, denoted by *).
3. **Context class**: Is the context class an in-memory or a persistent class.

<table>
<thead>
<tr>
<th>Context parameters</th>
<th>Pattern name</th>
</tr>
</thead>
<tbody>
<tr>
<td>context-class (in-memory) → * persistent-class</td>
<td>Persistent Collection Data Mapper (PCDM)</td>
</tr>
<tr>
<td>context-class (in-memory) 1 ← persistent-class</td>
<td>Memoization Proxy Data Mapper</td>
</tr>
<tr>
<td>context-class (in-memory) * ← persistent-class</td>
<td>Multi Memoization Proxy Data Mapper</td>
</tr>
<tr>
<td>context-class (persistent) → 1 persistent-class</td>
<td>Persistent to Persistent (P2P)</td>
</tr>
<tr>
<td>context-class (persistent) → * persistent-class</td>
<td>Persistent to Multi Persistent (P2MP)</td>
</tr>
</tbody>
</table>

Table 1 - Context Data Access Patterns

The context patterns extend the application of the PDM pattern with context considerations: They directly refer to or update the Data Access Layer classes created by the PDM pattern. Therefore, they require that the PDM pattern has been already applied. Each pattern is characterized by a context that singles out a persistent class and its interaction with another in-memory or persistent context class. The context of a pattern raises problems to which the pattern provides solutions.

**Persistent-Collection-Data-Mapper (PCDM):**

1. **Context**: Applies to an in-memory collection of objects of a persistent class, like the `scanSet` collection of `Message` objects in Figure 2.
2. **Problem**: The collection might become too big, although holding only lightweight proxy instances. It might be desirable to replace the proxy objects by their IDs. However, this raises another problem of proxy object deletion, since proxy objects might be deleted from memory, while still referenced via collections of IDs. Thus, losing the in-memory reference might initiate the deletion of the persistent object from storage.
3. **Solution**: The structure of the suggested solution is illustrated in Figure 4. It involves the addition of a context-specific collection class (`MessagesCollectionMapper`), that is responsible for loading and unloading proxy objects to memory per client request. The client class of the collection does not change as the context-specific collection class is hidden, being a subtype of a regular library collection class (e.g., `List`). The new collection is constructed by redirection of a factory method of the collection. The proxy object deletion problem is solved by extending the garbage collector to handle stored objects. A simple solution, demonstrated in Figure 4, involves reference count (but it does not cope with problems of reference cycles). The `Proxy` class is associated with a new Data Access Layer class named `TableManager`, which guards storage deletion of each persistent instance by tracking reference counts. The collection itself (i.e., `MessagesCollectionMapper`) can be made persistent by application of the PDM pattern to the collection.
Memoization-Proxy-Data-Mapper:

1. **Context**: A persistent class with an outgoing reference to a single in-memory class, like the `<Message, EncryptionKey>` context in Figure 2.
2. **Problem**: Maintaining the in-memory reference while objects are stored.
3. **Solution**: The structure of the suggested solution is described in Figure 5. The responsibility for maintaining the reference is imposed on the proxy object.

Multi-Memoization-Proxy-Data-Mapper:

This pattern is similar to the Memoization Proxy Data Mapper pattern. The only difference is that it handles a many cardinality in an outgoing reference like the `<Message,
UndoData association in Figure 2. The solution is similar to the previous pattern, and is skipped, for the sake of brevity.

**Persistent-To-Persistent (P2P):**

1. **Context:** A persistent class with a reference to a single different persistent class, like the <Message, Attachment> context in Figure 2.
2. **Problem:** The problem is how to consistently handle the reference between instances of two persistent classes, since the stored reference is duplicated in-memory. The latter is restored from, and should be kept consistent with, its Data layer origin.
3. **Solution:** The structure of the suggested solution is described in Figure 6. The solution suggests that the data mapper class of the persistent object (e.g., MessageDataMapper) receives the additional responsibility of keeping the in-storage reference updated at all time.

![Figure 6 - Structure introduced by the Persistent-To-Persistent pattern](image)

**Persistent-To-Multi-Persistent (P2MP):**

1. **Context:** A persistent class whose instances hold references to collections of instances of another persistent class (a 1:many cardinality), like the <MailServer, Mail> context in Figure 2.
2. **Problem:** The problem is how to consistently handle the reference between instances of two persistent classes. The problem arises since the collection is duplicated in memory, and has to be restored from, and kept consistent with, its Data layer origin.
3. **Solution:** The structure of the suggested solution is described in Figure 7. A new type of a persistent-collection is introduced, for handling the 1:many reference. The new collection stands in-between the two persistent classes and provides all collection services, as a regular in-memory collection. However, its services are based on information from the stored collection.

   In addition, the datamapper of the referencing persistent object (e.g., MailServerDataMapper) is responsible for keeping the in-storage and the in-memory relations synchronized and updated, whenever the referencing persistent object is created, loaded from storage, updated and deleted.

   Apart from the Persistent-Collection-Data-Mapper (PCDM) pattern, all patterns solve essential partial persistency problems that are not handled by the PDM pattern. The PCDM pattern is optional since the PDM pattern handles the problem of incoming references (i.e.,
clients of the persistent class), as it leaves the persistent class interface intact (making it abstract). Therefore, clients of the persistent class stay intact, meaning that there is no problem of incoming references. The DAL-insertion algorithm introduced in the next Section, allows for an optional application of this pattern.

![Diagram of Persistent-To-Multi-Persistent pattern](image)

Figure 7 - Structure introduced by the Persistent-To-Multi-Persistent pattern

4 Insertion of the Data Access Layer

The DAL patterns presented in the previous chapters provide solutions to local problems that result from partial persistency. Each pattern solves the problems it sets for itself, but neglects problems handled by other patterns. For example, the application of the Proxy-Data-Mapper pattern loses all outgoing references from the class marked as persistent (either into memory classes or into other persistent classes). Therefore, this pattern by itself does not preserve the system behavior and requires the "help" of other patterns. Consequently, the insertion of the Data Access Layer (DAL) must involve multiple patterns, and raises questions of pattern combination and of evaluation of the overall insertion procedure.

Pattern dependency constraints:

1. **PDM applies first**: Given a class that is marked as persistent, no context pattern can be applied to it, unless the PDM pattern was applied to it before (the context patterns require the existence of a proxy and a datamapper classes for the persistent class).

2. **Memoization excludes PCDM**: The Persistent Collection Data Mapper pattern should not be applied to a collection whose element class satisfies the context of one of the memoization patterns. For example, in the email application system described in Figure 2, PCDM cannot be applied to the ScaneSet collection of Message objects, because the memoization patterns apply to the contexts <Message, EncryptionKey> and <Message, UndoData>. This constraint is set because the PCDM pattern replaces the lightweight proxy representatives of stored objects by their IDs, thereby losing the rest of their structure (if exists). But, when the memoization patterns apply, the proxy objects include, besides the IDs, the outgoing references to the memory objects.

3. **P2MP excludes PCDM**: The PCDM pattern should not be applied in a context that satisfies the Persistent to Multi Persistent (P2MP) pattern (and therefore satisfies also the context requirements of PCDM). For example, in the email application system in Figure 2, PCDM should not be applied to the outgoingFolder collection of Message objects because the P2MP pattern applies to the context <MailServer, Message>. The reason for this constraint is that the P2MP pattern associates the client (e.g.,
MailServer) with a collection object (e.g., MessageCollectionMapperImp) that is associated with the storage collection (e.g., the collection of stored Message objects). If PCDM applies after P2MP, then the collection associated with the client looses the connection to the stored collection. If PCDM applies before P2MP, then the dangling collection created by PCDM is lost.

Algorithm 1 below is a non-deterministic DAL insertion algorithm that involves application of all DAL patterns. Non-determinism reflects decisions that have no impact on the final result (although they might affect performance and efficiency). Deterministic decisions result from the inter-pattern dependency constraints.

**Algorithm 1 DAL-Insertion algorithm**

procedure DAL-Insertion(PersistentClasses, PersistentCollections)

1. Apply Proxy-Data-Mapper to all PersistentClasses.
2. Non-deterministically apply the following patterns, in their appropriate contexts:
   a. Memoization-Proxy-Data-Mapper.
   b. Multi-Memoization-Proxy-Data-Mapper.
   c. Persistent-to-Persistent.
   d. Persistent-to-Multi-Persistent.
3. Optional: Apply Persistent-Collection-Data-Mapper to collections of persistent classes \( P \) such that:
   (1) \( P \) does not participate in a context of a memoization pattern (i.e., excluding all collections of classes handled in 2.a or 2.b.),
   (2) The collection does not fall in the context of the Persistent-to-Multi-Persistent pattern (i.e., excluding all PersistentCollections handled in 2.d.).

end procedure

Figure 8 demonstrates the application of the algorithm to the email system from Figure 2. The patterns require that collection classes are implemented using wrappers, and that all persistent classes and collections have factory methods. The newly created Data Access Layer classes are marked in gray color.

**Anytime variation of the DAL-insertion algorithm:** Replace the breadth-first pattern application by a depth-first one. That is, for every class marked as persistent apply first the PDM pattern, followed by the memoization patterns and the P2P and P2MP patterns in any order, with the PCDM applied optionally at the end. This variant has an anytime algorithm flavor because at any point of its application there is a set of classes that are marked persistent, and are already fully integrated with the Data layer.

**Correctness of the DAL-Insertion Algorithm**

Correctness requires the specification of properties that must be fulfilled by the algorithm under consideration. Overall, there are three major requirements: Partial persistency, transparent inter-layer integration and behavior preservation.

Partial persistency means that all and only instances of classes that are marked as persistent are stored in some durable storage. Transparent inter-layer integration means minimal changes to the existing layers. An acceptable integration might involve, at most, changes to the integrated classes (the classes marked as persistent) and to their domain layer clients.
Figure 8 - Application of the DAL-Insertion algorithm to the email system example

Behavior preservation is difficult to formalize. In simple formal computation models, it means tool equivalent, e.g., automata equivalence. But, in a complex enterprise system with multiple layers, and complex intra-layer structure, the notion of behavior preservation is still not well defined. Indeed, the refactoring literature includes many suggestions towards its formulation. In the lack of a well defined behavior preservation notion, a common approach is to replace it by several measurable observables. We suggest observables for preservation of structure, inter-object reference, object availability and method computation. Overall, we obtain six requirements that define the correctness of the algorithm.

Requirements for correct Domain - Data Layer integration:

1. **Partial persistency**: All and only instances of classes marked as Persistent are stored in durable storage.
2. **Transparent Domain-Data Layer integration**: Minimize Domain layer changes.
3. **Object structure preservation**:
   a. **Domain-Data Layer consistency**: In memory objects that duplicate storage objects have an identical value (state).
   b. **Intra-memory consistency**: In memory objects are not duplicated.
4. **Preservation of references between objects**:
   a. **Ingoing reference preservation**: All in-memory client references to objects of a persistent class are preserved.
   b. **Outgoing reference preservation**: All references from objects of a persistent class to objects of an in-memory class are preserved.
5. **Preservation of object availability**: Persistent objects are duplicated in-memory whenever a client requires them.
6. **Method computation preservation**: All methods are preserved. That is, all methods of the original system exist in the refactored system, and their computations might differ only by additional Data Layer accesses.

A DAL insertion algorithm is correct for Domain-Data integration if its application satisfies the above requirements. A set of DAL patterns is complete if there exists a correct DAL insertion algorithm that is based only on a combination of patterns from the set. A complete set of DAL patterns is minimal if the removal of a pattern from the set turns it incomplete.

**Claim 1.** The DAL-insertion algorithm is correct.
The claim holds since it can be shown that an application of the algorithm indeed satisfies all six requirements. In particular, the transparent integration requirement holds since the Domain layer stays intact. The duplication requirements are satisfied by the individual patterns; reference preservation is handled by the memoization patterns, and object availability is guaranteed by the proxy object structure and the tableManagers of persistent classes that guard against unintended deletion from storage. Method computations are preserved because the Domain layer stays intact, and the proxy objects wrap object creation, application, update, and deletion, with Data Layer access operations. Therefore, it is a correct algorithm for integration of the Domain-Data Layers.

Claim 2. The set of DAL patterns, without the Persistent Collection Data Mapper pattern, is complete and minimal.

The argumentation is immediate. The set is complete since the DAL-insertion algorithm in which the Persistent Collection Data Mapper pattern is optional, is correct. The set is minimal because each pattern solves a problem that is not solved by other patterns.

5 Discussion and Future Work

In this paper we introduced a set of independent Data Access Patterns that provide the missing link towards full automation of the Domain-Data layer interaction (without inheritance). The set of patterns is complete and minimal with respect to the task of Data Access Layer insertion. A correct DAL insertion algorithm was presented as well. The novelty of the suggested integration is that it leaves the Domain layer intact. The set of patterns includes an additional optional pattern that can optimize in-memory collections of persistent objects. Patterns for handling persistency requirements within a class hierarchy structure are still under development.

All patterns have been implemented as independent software refactorings. Current implementation is still restricted to an experimental PET tool that operates only on static class diagrams. Future development is planned for full application of the DAL-insertion algorithm, and its embedding in a tool for automatic integration of the Domain-Data layers.

Automatic integration requires coping with general problems of applications that support Domain-Data layers. The major problems that require consideration are recovery handling, support for transactions and query optimization, and the problem of reduced performance.

Another research direction involves the application of this approach for the automation of integration of the domain layer with networking services (i.e. web-services), that are functioning as data providers.

6 References

[4] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley, (1994).


