PATTERNS FOR AUTOMATIC INTEGRATION OF THE BUSINESS-DATA LAYERS IN ENTERPRISE-SYSTEMS

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE M.Sc DEGREE

By: Lior Limonad

June 2006
PATTERNS FOR AUTOMATIC INTEGRATION OF THE BUSINESS-DATA LAYERS IN ENTERPRISE-SYSTEMS

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE M.Sc DEGREE

by Lior Limonad
Supervised by: Prof. Mira Balaban

June 2006
This thesis is dedicated to my beloved grandmother Ita and my precious parents Cila and Jacob. Without their support, these lines would not have been written. I wish them all health and many more years of “strong living”!
Abstract

Nowadays enterprise systems are built in layers. The layered structure enables independent development of modules, coping with varied platforms and technologies, and stability with respect to rapid change. Nonetheless, organizing large-scale systems into discrete layers of distinct, related responsibilities, with a clean, cohesive separation of concerns, is still is a great challenge for developers.

One common challenge of layers integration is between the Domain and Data layers of the application involved with persistency insertion step. The Domain-Data layers integration reflects storage concerns, implying complex interaction between the layers, due to information duplication and overlapping responsibilities.

Existing technologies offer partial solutions for persistency insertion. Nevertheless, the developer still has to design the concrete ties between the layers, which still requires writing a large amount of annoying, bug-infected code.

This work suggests a set of Data Access Patterns as the solution that ties the missing link between the Domain and Persistent layers. All patterns rely only on local information in the Domain layer. Each pattern is realized as refactorings over a given application that yields some Data-Access layer constructs. The Domain layer stays intact.

As an overall solution for layers integration, this work also suggest a combined algorithm for patterns application. The suggested algorithm satisfies the Domain-Data integration requirements. Inclusively, this thesis provide a solution for the missing link for achieving full automation of the Domain-Data layers integration. An experimental tool that was implemented is also presented.
Acknowledgments

My deepest gratitude goes to my academic adviser, Prof. Mira Balaban who had agreed to share with me all of her invaluable wisdom, knowledge and gained experience. I will sure miss all our thrilling conversations from which I have always earned new insights. No matter what time and how distressed she was, I was always accepted with fondness. Guiding me through all obscurities, this research would not have been possible without her support.

A special thanks goes to Maria Itkin and Edward Berezitsky for their efforts in the development of PET, adhering to all my capricious, and for Dr. Arnon Sturm who had devotedly advised in their progress.

In addition, I would like to thank Dr. Michael Elhadad and Dr. Mayer Goldberg who had provided me, out of their free will, with highly effective criticism regarding this work.

It is a pleasure to express my thanks to my boss, Mendi Hanner, who had provided me with a perfect working environment and warmth attitude.

Special thanks to all my closest friends: Assi, Alon & Tali and Erez & Maya. I would like to thank them all for showing me there are excitements in life other then “persistency”.

Finally, I thank my cherished family: My grandmother Ita, my parents Cila & Jacob, my brother Ofer and his girlfriend Natali and my loving girlfriend Sigal. Their support was of vital importance for my personal and academic advancement.
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1.1</td>
<td>Persistent Collection Data Mapper - Detailed Description</td>
<td>48</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Persistent Collection Data Mapper - Related Patterns</td>
<td>50</td>
</tr>
<tr>
<td>4.2</td>
<td>Memoization Proxy Data Mapper</td>
<td>53</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Memoization Proxy Data Mapper - Detailed Description</td>
<td>54</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Memoization Proxy Data Mapper - Related Patterns</td>
<td>56</td>
</tr>
<tr>
<td>4.3</td>
<td>Multi Memoization Proxy Data Mapper</td>
<td>58</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Multi Memoization Proxy Data Mapper - Detailed Description</td>
<td>59</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Multi Memoization Proxy Data Mapper - Related Patterns</td>
<td>61</td>
</tr>
<tr>
<td>4.4</td>
<td>Persistent to Persistent</td>
<td>62</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Persistent to Persistent - Detailed Description</td>
<td>66</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Persistent to Persistent - Related Patterns</td>
<td>68</td>
</tr>
<tr>
<td>4.5</td>
<td>Persistent to Multi Persistent</td>
<td>69</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Persistent to Multi Persistent - Detailed Description</td>
<td>73</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Persistent to Multi Persistent pattern - Related Patterns</td>
<td>74</td>
</tr>
<tr>
<td>5</td>
<td>Towards Automatic Insertion of the Data Access Layer</td>
<td>77</td>
</tr>
<tr>
<td>5.1</td>
<td>Data Access Layer insertion correctness</td>
<td>77</td>
</tr>
<tr>
<td>5.2</td>
<td>Dal-Insertion Algorithm</td>
<td>81</td>
</tr>
<tr>
<td>5.3</td>
<td>Overall Application Example</td>
<td>83</td>
</tr>
<tr>
<td>6</td>
<td>Patterns Execution Tool (PET) implementation</td>
<td>89</td>
</tr>
<tr>
<td>7</td>
<td>Conclusions and Future work</td>
<td>94</td>
</tr>
<tr>
<td>7.1</td>
<td>Recovery process</td>
<td>94</td>
</tr>
<tr>
<td>7.2</td>
<td>Transactions</td>
<td>95</td>
</tr>
<tr>
<td>7.3</td>
<td>Performance</td>
<td>96</td>
</tr>
<tr>
<td>7.4</td>
<td>Future Work</td>
<td>97</td>
</tr>
<tr>
<td>8</td>
<td>Appendix A - Transformation Building Blocks</td>
<td>100</td>
</tr>
<tr>
<td>8.1</td>
<td>DAL specific transformations</td>
<td>101</td>
</tr>
<tr>
<td>8.2</td>
<td>General purpose transformations</td>
<td>135</td>
</tr>
<tr>
<td>8.3</td>
<td>Primitive conditional transformations</td>
<td>145</td>
</tr>
<tr>
<td>9</td>
<td>Appendix B - Smart Unloading Mechanism</td>
<td>148</td>
</tr>
</tbody>
</table>
# List of Figures

1.1 Partial duplication complexities ................................................. 2  
1.2 A mixture of in-memory and persistent classes ................................. 4  
2.1 The structure of a Decorator .................................................... 13  
2.2 The structure of a Proxy .......................................................... 14  
2.3 The structure of an Active Record .............................................. 15  
2.4 The structure of a Data Mapper .................................................. 16  
2.5 Data Mapper Example: Retrieving Data from a Database ...................... 16  
2.6 Identity map object retrieval ..................................................... 17  
2.7 OO - Relational GUI Based Mapping definition ................................ 20  
2.8 A typical XML descriptor of an Enterprise-Application ....................... 20  
3.1 An in-memory Message class marked as persistent ............................ 22  
3.2 A possible model for an e-Mail application ................................... 23  
3.3 Using Fowler’s Active Record for the described case ........................ 25  
3.4 Using Fowler’s Data Mapper for the described case .......................... 26  
3.5 Updating the attachment of an existing Message ................................ 27  
3.6 A MessageDataMapper that inherits from Message ............................ 30  
3.7 Combining the Mapper and Proxy concepts together .......................... 31  
3.8 Proxy Data Mapper - Usage Example .......................................... 34  
3.9 Example for dbSchemaElement ................................................... 35  
4.1 Persistent Collection Data Mapper - Context .................................... 45  
4.2 Persistent Collection Data Mapper - Analysis model context ................ 45  
4.3 Structure introduced by the Persistent Collection Data Mapper ............. 51  
4.4 Persistent Collection Data Mapper - Extended Action Result ................ 52  
4.5 Memoization Proxy Data Mapper - Context .................................... 53  
4.6 Structure introduced by the Memoization Proxy Data Mapper ............... 57
4.7 Multi-Memoization Proxy-Data-Mapper - Context ........................................ 58
4.8 Structure introduced by the Multi-Memoization Proxy-Data-Mapper ................. 62
4.9 Persistent-to-Persistent - Context ............................................................... 62
4.10 Load implementation of the persistent class mapper .................................... 65
4.11 Structure introduced by the Persistent-to-Persistent ................................ 68
4.12 Persistent-to-Multi-Persistent - Context .................................................... 69
4.13 1:M Implementation in a Relational database ............................................. 72
4.14 Structure introduced by the Persistent-to-Multi-Persistent ............................. 75

5.1 Application Example - Input system ............................................................ 83
5.2 Application Example - Proxy Data Mapper result ......................................... 84
5.3 Application Example - Memoization Proxy Data Mapper result ....................... 85
5.4 Application Example - Multi Memoization Proxy Data Mapper pattern result ... 86
5.5 Application Example - Persistent to Persistent result .................................... 87
5.6 Application Example - Persistent to Multi Persistent result ............................ 87
5.7 Application Example - Final result .............................................................. 88

6.1 PET input and output .................................................................................... 90
6.2 PET - example of usage .............................................................................. 90
6.3 PET components ......................................................................................... 93

7.1 Hibernate transaction handling .................................................................... 96
List of Tables

4.1 Context Data Access Patterns ............................................. 43
Chapter 1

Introduction

“I never think of the future - it comes soon enough.”

Albert Einstein, physicist (1879 - 1955)

Over the past decade, systems infrastructures have extremely evolved to keep up the pace of changing environment and globalization. No doubt is that many large companies like IBM, Microsoft, Oracle and Sun, have re-prioritize their agenda to meet with the needs of enterprise businesses. These needs had set up new challenges to be achieved. New concepts such as Middleware and Service-Oriented Architectures have been introduced, and are today within central concerns of stakeholders in software community.

Keeping up with the pace, modern software development methods today grasp changing requirements as the norm. Nowadays methodologies, such as Extreme-Programming [2], Unified-Process [14] and Scrum [30], accompanied by supporting methods like Test-Driven-Development [3], Refactoring [10] and Aspect-Oriented-Programming [15], all adhere to the notion of agility [18]. That is a development process which is short timeboxed, highly adaptive and driven by change. The common belief is that traditional approaches such as the waterfall model and the spiral model [31], which are based on slow phased and strictly sequential philosophy that cannot embrace change, are sooner or later doomed to extinct.
Among all software development approaches today, one very popular is the Model-Driven-Approach (MDA) [20]. According to the MDA spirit, application development is perceived as an iterative process of creating different models from different aspects and layers of the problem space, and then automatically weaving them into the form of an implementation.

Yet, current status is that models have loose semantics, they are loosely related to code and quickly become irrelevant as a system is developed. Moreover, the layers in a system adhere to different responsibilities and roles. Thus, their weaving process must provide the missing links that may integrate them together. This occasionally requires the population of new inter-relating layers and forces the developer to manually write a tedious code. Hence, weaving the models together is still a major weak point in the development process.

One common challenge of layers integration is between the domain and data layers of the application involved with persistency insertion step. Although both layers intersect by their unified semantics, they are detached by their physical storage location.

![Figure 1.1: Partial duplication complexities](image)

Most enterprise systems present partial persistency, that is, some of the objects in the system are persistent and the other are in-memory objects. Thus, Domain-Data layers integration entails partial duplication of persistent elements between the layers. The partial duplication gives rise to several complexities (shown in Figure 1.1):

1. **Consistency**: Duplicated objects between the layers (i.e. Element1 and Element3) and duplicated objects within the same layer (i.e. Element3 within the Domain layer) must be coherent,
reflecting the same information and structure at any given time.

2. **Mapping:** *Mapping* defines the linkage between elements of two layers on a semantic basis. Since the layers had evolved independently, through maturity each had embraced different constructs to support its physical environment. This parallel evolutionary process had led to the widely discussed *mismatch*\(^{[32, 29, 16, 24, 12]}\) between two popular realizations: *object* and *relational* models. This mismatch had motivated the development of two-way *object-relational* mapping theories \(^{[6, 17, 1]}\).

3. **Behavior preserving:** All elements behavior after *partial duplication* is introduced should be equivalent to their behavior prior to the insertion of persistency considerations. Duplicated objects must preserve all of their dynamic structure and service realizations.

4. **Availability:** All duplicated objects must expose the same accessibility level before and after persistency insertion. That is, each persistent object must be accessible to all other objects interacting with it.

5. **Mixture:** Since there is a mixture of in-memory and persistent objects, all ingoing and outgoing links, into and from each persistent objects (marked with a dashed line in the figure) must be handled whenever it is being duplicated and restored.

Existing technologies offer partial solutions for persistency insertion. Several tools such as Java’s Hibernate and Firestorm assist the developer in creating concrete mappings between elements of two layers and provide convenient abstractions on top of concrete database systems. Standards like Sun’s JDBC, JDO, DAO and J2EE and their corresponding implementations in industry do provide various means for handling duplications and for connecting an application to one specific storage system or another.

Nonetheless, the mentioned *mixture* still entails the developer to provide the concrete ties between the layers, taking care of all persistent to in-memory possible interactions, accounting for collections of persistent objects and handling of inconsistent duplication problems.
For example, Figure 1.2 presents a possible model for an email system that has been created to realize the use-case - “scan all messages for virus infection”. The implementation requires that for each scanning procedure, a new VirusScan object is constructed to iterate over all Message objects and their Attachments. All messages and attachments in the system are meant to be stored. Therefore, both classes Message and Attachment are marked persistent.

![Figure 1.2: A mixture of in-memory and persistent classes](image)

This model exposes a typical mixture of in-memory and persistent classes. There are two different types of in-memory to persistent interactions illustrated in this model. One is between the VirusScan class and the Message class (in-memory to persistent) and the other between the Message class and the Attachment class (persistent to persistent). The developer must provide an implementation that not only satisfies the local persistency requirement for each class marked persistent, but also one that resolves the interactions between them. This realization entails answering many confusing questions such as:

- How to make sure that each Message object is duplicated to memory from storage when required by its VirusScan client object?

- How to reconnect the loaded copy to its corresponding attachments as appears in storage?

- What impact does it have on the VirusScan class implementation? On other classes in this model?

- Whenever an in-memory copy of Message has changed, how to preserve its consistency with its corresponding copy in storage?

- Whenever an Attachment object is deleted by the scanner as being virus-infected, how is it deleted from storage?
Which class has the responsibility to handle that? all other persistency related problems?

All and many other confusing questions are to be answered to achieve an acceptable implementation. Writing the code that corresponds to all interaction types, requires the analysis of each interaction, taking care of ingoing and outgoing links, into and from persistent objects. An annoying procedure which might lead to bug-infected code. Full automation can be a great improvement and goes along with the Model Driven Approach.

This work suggests a set of Data Access Patterns (DAL) as the solution that ties the missing link between the Domain and Persistent layers. The patterns include identification of all local incidents to be handled by the application. This include classes marked as persistent and their interactions with other classes. Each incident has its solution in a corresponding pattern.

All DAL patterns have the following characterizations:

- The solution is completely algorithmic.
- Input for each pattern requires only the marking of persistent classes and collection of these classes. No global semantic analysis of class interactions is required (i.e. data flow graph).
- All patterns leave the domain layer intact.
- All new constructs are part of the Data-Access layer.

All patterns in this work are defined as a combination of smaller refactorings. All patterns have been implemented using simulation of object oriented meta-model. The patterns were applied with respect to an example e-mail system. Furthermore, an experimental tool that applies these patterns on a given class-diagram schema is implemented.

Each pattern solves local problems and does not provide a global solution to the problem of partial persistency. Therefore, in order to solve the problem, the patterns should be combined. Therefore, this work also suggests a DAL-Insertion algorithm as the combined procedure for patterns application. As its input, the Dal-Insertion algorithm requires only the following local information: the system to work upon and marking of all persistent classes and of all persistent collections.
We claim that the *Dal-Insertion* algorithm solves the problem of *partial persistency*. This entails the definition of *partial persistency* using quantifiable criteria. Intuitively, the realization of *partial persistency* requires:

1. Persistency of all and only classes marked persistent.
2. Domain-Data layer integration.

Since *behavior preservation* is hard to quantify, we introduce measurable observables that sum up into our definition of *behavior preservation*. Each pattern is correct with respect to its own observables (problems).

The *Dal-Insertion* algorithm that combines all patterns fulfills all *partial persistency* observables and, therefore, is claimed to be correct. Moreover, we show that the *Dal-Insertion* algorithm is minimal in the sense that it imposes only minimal requirements for satisfying all observables.

Inclusively, this thesis provides a solution for the missing link for achieving full automation of the *Business-Data* layers interaction. Given the above algorithm, the automatic integration of *Domain-Data* layers would take the following steps:

1. Domain layer marking of:
   
   (a) Classes to be persistent.
   
   (b) Persistent classes collections.

2. Data layer:

   (a) Mapping specification between domain-layer classes and storage elements (i.e. Tables).
   
   (b) Access specifications (i.e. Driver).

3. Application of DAL-Algorithm over all given input.
The contribution of this work is in closing the missing link that ties the interactions of Domain and Persistent layer. Moreover, the provided algorithm can be applied at any stage of system’s life-cycle. Since all patterns leave the domain layer intact, the algorithm is oblivious to the stage in the development it is applied in.

The structure of this thesis consists of seven parts:

Chapter 2: Reviews all background materials that was required to support the insights achieved by this work. This includes the understanding of nowadays enterprise system architecture, current means to reshape it and status of current related technologies.

Chapter 3: Discusses the ProxyDataMapper (PDM) pattern that serves as the core pattern for all other DAL patterns. The beginning of this chapter details its evolutionary process till problem satisfaction accompanied by an example. The remaining of the chapter details the algorithm for applying the core pattern.

Chapter 4: Details all Data-Access-Patterns that are complimentary to the application of the core pattern. Each pattern definition has its problems discussion, application input, application pre-condition and post-condition and a detailed explanation for its application procedure.

Chapter 5: Introduces the Data-Access-Layer Insertion (DAL-Insertion) algorithm as the method for the combined application procedure of all patterns and claims for its correctness. An application example of the algorithm is given as well.

Chapter 6: Summarizes the functionality and architecture of the experimental PET tool that has been developed as a spike for this work.

Chapter 7: Discusses several aspects that intersect with concern of this work and suggests further expansions to this framework.

Chapter 8: Concretely details the realizations for all patterns transformations accompanied by code examples.

Chapter 9: Details some code related to an example given in this work.
Chapter 2

Background

This chapter details several basic notions that serve as the foundation on which the framework, presented in later chapters, is built upon. The scope of the first section is static and dynamic aspects of current enterprise systems architecture. The second section presents related means for designing and improving the architecture of such systems. The last section introduces current technologies and tools that support several concerns related to enterprise systems and to the scope of this work.

2.1 Architecture of Enterprise Systems

2.1.1 Layering

Modern enterprise systems are built in layers [9, 19]. The layered structure enables independent development of the modules, coping with varied platforms and technologies, and stability with respect to rapid changes. All layers need to co-exist and co-evolve in integrated complex systems. While small systems can be two layered, following the classical client-server approach, large systems must adopt a richer structure that includes, at least, a presentation, a domain and a data (persistency) layer.

- **Presentation layer:** contains all software parts needed to handle interaction between the user and the system. This includes parts such as graphical user’s interface components, web interfaces, client side consoles, automatic voice answering, web services etc.
2.1. Architecture of Enterprise Systems

- **Domain layer:** The domain layer is probably the heart of any complex system. It encapsulates all software parts needed to fulfill activities required by the domain it is located in. This layer is also referred to as *business logic* layer. The layer may include parts that are responsible for calculations, validation of data, allocation of resources, image processing, policy enforcement, voice analyzing and so forth.

- **Data layer:** Encapsulates all software parts needed to communicate with resources located outside the system. This may include: interacting with communication devices, network access to other systems, activation of reporting devices like plotters, and most relevant to this work, access to different kinds of database platforms.

A good layering architecture supports the following features:

1. Enables inter-layer communication.
2. Enables information duplication between layers, e.g., *domain – data* layers.
3. The structure of one layer does not affect that of other layers.
4. Insertion of a new layer does not spoil existing ones.

However, organizing large-scale systems into discrete layers of distinct, related responsibilities, with a clean, cohesive separation of concerns, has been and still is a great challenge for developers. Regrettably, the fame is bounded with a lot of “pains”[5].

One most common challenge of layer integration is between the *domain* and *data* layers. Although both layers intersect by their unified semantics, they are detached by their physical storage location. Therefore, their incorporation requires not only technological support but also manual integration that specifies the linkage between elements of two layers.

Current solutions provide only partial solution that involves all complexities related to persistent and non-persistent object interactions. This work suggests a set of *Data Access Patterns* (DAL) as the solution that ties the missing link between the *Domain* and *Persistent* layers. Each pattern yields
new Data-Access layer constructs that solve a concrete interaction between duplicated objects in the Domain layer.

### 2.1.2 Model-Driven Approach

The rising Model-Driven Architecture (MDA) approach [21] presents additional challenges of automatic software development. Not only the layers discussed in section 2.1.1 have to be automatically developed, starting from software models, but they also need to be automatically integrated.

MDA is a standard concept from the Object Management Group (OMG) that allows developers to link object models together to build complete systems. MDA prevents design decisions from being intertwined with the application and keeps it independent of its implementation. The main concept of MDA is to raise, once again, the level of abstraction [20] together with supporting tools to map from one layer to the next automatically.

The central building block of the MDA is the Model. Each Model consists of sets of elements that describe some physical, abstract, or hypothetical reality. Dealing with models, the development process is accelerated since only matters within the same level of abstraction are discussed within the context of each model. Issues and concerns from different levels are now clearly separated.

Each model is expressed using a combination of text and several complementary and interrelated diagrams. Today, the most common modeling language used is UML. However, since the spirit of MDA is to encourage further evolution of models and their supporting tools, alternate domain-specific modeling languages are now likely to be an alternative.

Hence the development process according to MDA is conceived as an iterative process of creating different models at different levels of abstraction and then linking them together to automatically form an implementation. The idea is that linkage between models is achieved by their inter relationship mapping definitions and markings. Mappings define the semantic binding functions between model elements and markings define model extensions that capture information required for model transformation.

Adhering to the MDA spirit, the framework introduces by this work deals with the automatic
insertion of a data access layer that is responsible for the connection between the domain and the data layers. The solution illustrated in this framework for persistency realization, is to be conceived as the definition of mappings (termed patterns) to support automatic model transformation between a domain that does not embed persistency consideration into a domain that does.

2.2 Design of Enterprise Systems

2.2.1 Design Patterns

Design patterns are recurring solutions for recurring problems in different areas of our life. They have been found most useful in the software community, especially in solving problems for which no specific solution algorithm is available. Design patterns are convenient way of reusing object oriented code between projects and between programmers [7, 16]. The idea is to catalog common interactions between objects that programmers have often found useful.

The first breakthrough of software design patterns was in 1995 when Gamma, Helm, Johnson and Vlissides published their book - *Design Patterns - Elements of Reusable Software*, later referred to as the *Gang of Four* (GoF) book [11]. The book summarized a collection of 23 common and useful patterns. Since this first breakthrough and till today, the importance of design patterns has become clear in the software development community. Not only that patterns are common tools to improve design, but also they serve as a powerful means for communication in the software community.

Today, software patterns come in many flavors covering various aspects and concerns in the development activity. Apart from traditional design patterns, the diversity of patterns includes but is not limited to analysis patterns that focus on organizational, social and economical aspects of a system, enterprise patterns that cope with recurring problems in large-scale systems and more domain specific patterns such as data access patterns and interaction patterns.

Each pattern proposes an optional solution for a known problem. Usually the solution is outlined in general so it may be applied in different circumstances. Therefore, when applying the solution
offered by the pattern, it has to be tuned to specifically fit into the problem domain. Moreover, although most traditional patterns are relatively independent, the application of one pattern may lead to the application of another. Sometimes there might even be a trade-off between several patterns that seem relevant to the context of a problem.

Most design patterns have a defined structure. When the purpose of design patterns is to solve common programming issues, they usually share the following structure:

- The problem definition.
- Considerations for the problem or the context of the problem.
- A general solution - usually using the notation of UML today.
- Sample code for the solution.
- Good and bad consequences of the pattern.
- Related patterns.
- Code examples.

In most cases the designer’s judgment is required for the identification of a pattern as applicable. Deep introspection into the problem and identification of external factors that might have an impact on the concrete solution that has to be tailored may be involved. In such cases, automatic pattern application tools are useless. Nonetheless, in many cases, when the context of the pattern has been identified as an input, enough information can be automatically gathered to customize the appropriate solution.

In this work, the comprehensive solution is composed of a set of patterns. All patterns are automatically applicable based on the marking of persistent elements. Each pattern solution has been strictly investigated and analyzed in the shade of existing patterns from various domains. (i.e. storage [22] and enterprise systems [9]). Corresponding to the problem of persistency handling, the solution achieved is related to several common patterns. The following list details several pattern related to the scope of this work:
1. Two relevant structural GoF patterns [11]:

(a) **Decorator**: Also known as the *wrappers* pattern. Its main intent is to add responsibilities to individual objects. Responsibilities can be added and removed at run-time simply by attaching and detaching decorator objects. Nonetheless, adding responsibilities is transparent since it do not affect other objects.

![Figure 2.1: The structure of a Decorator](image)

As illustrated by the GOF, Figure 2.1 shows the basic structure of a Decorator. Whenever a new functionality is to be added, a new concrete class (e.g. `ConcreteDecoratorB`) is introduced as a subtype of the abstract decorator. A chain of decorator objects is detached to the concrete component. That way, the client is not affected and responsibilities may be added or removed dynamically by adding and removing concrete decorator objects.

Referring to the problem of persistency handling addressed by this work, the first tendency was to apply the Decorator pattern as the local solution for each class that is marked as persistent. Relating to persistency services as to an extra responsibility to be added, the decorator pattern seemed to be the appropriate solution.

However, drilling down into the essence of persistency services, no extra functionality is to be added to any object of the persistent class. Except for the need to make it accessible
to other objects in the application. The notion of each class that is marked persistent
doesn’t change as the result of persistency requirements. Therefore, as detailed bellow,
the solution offered by the proxy pattern seems to be more appropriate.

(b) Proxy: The solution by this pattern provides a surrogate or placeholder for another
object to control access to it. The proxy pattern is applicable in various cases. For
example, when the creation of the object on demand is an expensive procedure and
should be avoided until it is actually needed to be used.

In the context of this work, the proxy pattern is applicable as a remote proxy. The meaning
is that the proxy provides a local representative for an object in a different address
space. In this case, to an object in storage. As mentioned above, the main difference
between proxy and decorator is in its intent, which implies a different structure. Decor-
ator adds functionality to its subject while proxy main purpose is to make its subject
available or unavailable. Therefore, decorators can be nested, while a proxy cannot.

The structure of the proxy pattern is illustrated in Figure 2.2. Whenever the client re-
quires acting upon the RealSubject that resides in a different address space, the Proxy
forwards the requests to it. Using this structure, the fact that an object resides in a
different address space is hidden from its client.

![Figure 2.2: The structure of a Proxy](image)

2. Fowler’s Data-Access patterns [9]:
(a) **Active Record Pattern:** This pattern overloads the class of persistent objects with all data access logic. The pattern is termed *active-record* since each object corresponds to a single row in a database table. As shown in Figure 2.3, each *active-record* class is responsible for saving and loading to the database, and also any domain logic that acts upon the data (*insert*, *update*, and *get* implementations).

![Figure 2.3: The structure of an Active Record](image)

The *Active Record* pattern may be considered when the domain model is not too complex. Since the *Active Record* overloads the domain class with all persistency services, if the business logic is complex, mapping persistent objects to storage schema gets the *Active Record* class to be very messy. Another drawback of using the *Active Record* is that it couples the domain design to the database design.

(b) **Data Mapper Pattern:** This pattern is the basic pattern, on top of which the patterns suggested by this work are built. The pattern is related to the realization of a single class that is marked as persistent, meaning that its instances should be stored in a durable storage. The Data Mapper (shown in Figure 2.4) is a layer of software that acts as a mediator between the in-memory objects and the database, keeping them independent of each other and the mapper itself. A new mapper class is created for each domain class that is chosen to be persistent. Using the Data Mapper pattern, the in-memory objects do not need knowledge that there’s even a database present, no SQL interface code, and certainly no knowledge of the database schema.

As illustrated in Figure 2.5, whenever a client needs to access some object of the persistent class, it addresses the mapper corresponding to the persistent class with the id
that is used to identify the instance to be retrieved. The mapper is not responsible for keeping identities of retrieved objects. This implies that the identity must be forwarded for each retrieval invocation by the client. The responsibility for handling all persistency services and caching mechanism is imposed on the mapper.

(c) **Identity map:** This pattern ensures that each object is loaded into memory at most once, by keeping every loaded object in a map and lookup objects using the map when referring to them.

In the most casual case, each persistent class has a single corresponding map in memory. However, this might be awkward when inheritance is involved, since each lookup needs to know to look in several polymorphic maps. In that case, a single map corresponding
to the whole inheritance tree is preferred.

The identity map object may be accessed across the application by attaching it to a registry object [9]. The registry object is a global object that exposes services that are accessible across an entire scope of the application (i.e. using static methods).

Figure 2.6 shows a simple example for using the identity map pattern for object retrieval. The identity map object is accessed through a finder (registry object). Whenever an object has been already loaded into the map, the finder will retrieve it directly from the map the next time it is requested and won’t load it again. This way, duplication is avoided.

![Identity map object retrieval diagram](image)

Figure 2.6: Identity map object retrieval

### 2.2.2 Refactoring

Refactoring offers the means for improving the code in existing systems. For example, the suggestions offered by design patterns can be imposed using refactoring. Refactoring is the process of changing a software system in such a way that it does not alter the functionality of the code, while improving its structure. Refactoring is motivated by the urge to improve design and make software easier to understand, maintain, change and reuse. Having refactoring reduces the amount of up-front design invested in a system. Nowadays, refactoring also offers a comprehensive set of tools to re-shape and improve the design of a system.
Refactoring is perceived as a *transformation-rule* that has a *precondition*, an *action* and sometimes a *postcondition*. All conditions and actions may be defined formally or in a more relaxed verbal explanation. Nevertheless, there is always some rational that motivates the execution of a refactoring.

For example, one very common refactoring is *addParameter* [10]. The motivation of this refactoring is given by the need to change a method, and the change requires information that wasn’t passed before. Therefore, a possible solution is to add an extra parameter to that method. In this example, the refactoring action appears as a set of step-by-step description of how to carry out the refactoring. For example, to apply the *addParameter* refactoring, the system developer should execute the following instructions:

1. Check to see whether this method signature is implemented by a superclass or subclass. If it is, carry out these steps for each implementation.

2. Declare a new method with the added parameter. Copy the old body of code over to the new method.

3. Compile.

4. Change the body of the old method so that it calls the new one.

5. Find all references to the old method and change them to refer to the new one. Compile and test after each change.

6. Remove the old method.

7. Compile and test.

Refactoring focus on various aspects of an existing system: packaging, architecture, styling, responsibilities, organization, simplicity and so forth. Some refactoring are already integrated into development tools, easing the development process and supporting all modern development approaches.
Related to the scope of this work, the suggested patterns can be viewed as software refactoring. They are implemented by breaking them into separate conditional transformations (not necessarily refactorings, as they do not necessarily preserve behavior). The transformations are described in the Appendix A (Section 8).

2.3 Current Technologies

Industry today offers a variety of standards and frameworks for the persistent layer. Standards like Java 2 Enterprise Edition (J2EE) [13, 26], JDO (Java Data Objects) and DAO (Data Access Objects) [8, 27] have been introduced to the market. Most standards are accompanied by industrial implementations such as IBM’s Websphere, Oracle’s Application Server, BEA’s Weblogic, JBoss.org and several persistent specific tools such as Hibernate and FireStorm.

Most of the tools share the following requirements:

- **Mapping specification:** There are many theories about mapping between the data and persistent domains, going both ways: forward from the OO model to the Relational model [21] and backwards (Reverse Engineering [23]). An intensive work was done and published in this field by Michael Blaha and William Premerlani [3]. All of the suggested methods for integration give very useful patterns, how to overcome the differences between these two models.

  Some tools like FireStorm implement mapping theories to achieve automatic generation of mapping specifications and for the persistent layer insertion. Other tools, offer GUI based environments for gathering mapping definitions manually, as shown in Figure 2.7. Using these tools do not solve all the aspects that are related to persistent layer insertion. Interactions between persistent and non-persistent objects and storage schema that are not isomorphic to the application structure, are all out of the scope of current solutions.

- **Meta-Data specification:** Usually referred to as application-descriptors and written in XML files format. These specifications specify various containment configurations specifying how
the application is driven within the environment that hosts it. This include various definitions for: transaction handling, security, caching, performance, availability and database access specification.

Adjustments to standards: Enforce strict adaptation to technology related interfaces and standards (e.g. Home/Remote interfaces when running within J2EE environment). Implies ad-hoc rewriting of persistent and non-persistent application classes.

Environment Setup and Tuning: Many tools require setup for complex, resource consuming hosting environment. Related to complex operational systems, this task is usually handled by trained and skillful manpower as it requires much experience for achieving performance satisfaction.

Indeed using current solutions provide enhanced capabilities to support nowadays rapid development methods. However, existing technologies offer only partial solutions for persistency insertion. Regarding to all current technologies, yet, there is no automated solution for integrating a
domain and a persistent layer, when a significant mixture of persistent and in-memory objects is required.

Most running application requires that all types are recognized by the application, but only some of the types, those marked as persistent, actually duplicate persistent layer data. This exposes a major weak point of current technologies, that is, the gentle boundary that links between types that are to be persistent and types that are not. Some existing solutions enforce the duplication of all types in order not to deal with the partial persistency problem (i.e. unintended object serialization). Others just ignore it, binding the programmer to manually tie between the layers by writing tedious code.

In this work a set of Data Access Patterns is introduced (Chapter 4) to bridge the gap of the missing link. All patterns rely on a core pattern termed Proxy-Data-Mapper which is presented first (Chapter 3). The application of each pattern yields some Data Access layer constructs, and relies on semantic analysis of class interactions in the Domain layer. Moreover, all patterns are Domain layer transparent, i.e., their application leaves the Domain layer intact.

We claim that by applying these patterns as a complementary method to current solutions for persistency insertion, full automation of the Business-Data layers may be archived.
Chapter 3

Core Pattern - Proxy Data Mapper

3.1 Pattern condition

A class that is marked as persistent therefore all of its objects has to be stored within a different later.

![Figure 3.1: An in-memory Message class marked as persistent](image)

3.2 The Problem

Assume a model for an e-mail client application as the one illustrated in Figure 3.2. The model contains several possible components that might compose an e-mail system. To illustrate the problem,
mainly focus on the class *Message* that represents an e-mail message that is being edited.

During some phase in a system’s life cycle, the designer or developer may decide to have *Message* class defined as persistent. This is due to the requirement to preserve the state of each message object while it is being edited. Other classes may be marked persistent as well due to various considerations, such as: back up, memory resources limit, logging, and continuity of execution.

As the result of persistency marking, the model reflects several classes that are marked persistent and others that remain as plain in-memory classes with a diversity of association types connecting these classes. For example, the persistent *Message* class is associated with the in-memory *EncryptionKey* class through and outgoing association and with the persistent *MailServer* class through an incoming association.

Altogether, disregarding this mixture of classes and their interrelationships, persistency requirements must be satisfied with respect to all classes marked as persistent in the system.

![Diagram](image)

Figure 3.2: A possible model for an e-Mail application

The problem is to enable persistent storage of the objects of the persistently marked class, henceforth “the persistent class”, while preserving the system’s behavior. The solution suggested below has the following properties (requirements):
1. **Persistent objects**: All instances of the *persistent class* are stored within some durable storage, e.g. a relational database.

2. **Transparent persistency insertion**: All in-memory clients related with the *persistent class* are oblivious to its persistency. Whether an instance of the *persistent class* is either memory-resident or stored in some storage should have no effect on any of its clients.

3. **Transparent persistency insertion**: The persistent class is not overloaded with storage information and access services.

   For conceptual reasons, each class should be a simple direct abstraction of what it is meant to be representing, therefore we don’t want to spoil the Business-Layer with persistency considerations. Moreover, for the simplicity of automation all persistency handling should be handled by newly created Data-Access-Layer classes.

4. **In-memory presence of stored objects**: The persistent class objects exist in memory whenever needed, exposing all their services and interrelationships with other instances. Whenever a persistent class object is not required, it should be released from being memory resident.

5. **In-memory-Storage consistency**: Persistent class objects that exist both in-memory and in-storage are consistent. In a default setting, the storage should reflect the same state of an in-memory persistent instance whenever it is created, updated or removed by the application.

6. **No in-memory duplication**: There is no in-memory duplicated copies of persistent class objects. Corresponding to each layer (Business and Data) only a single instance of the persistent instance may exists in it. Once duplicated in memory, inconsistencies between storage and memory are unavoidable, since the in-memory copies are independent of each other, and their updates cannot be synchronized.

The properties are necessary for guaranteeing that the pattern is behavior preserving, and that the combined application of all patterns (using DAL-Insertion algorithm in chapter 5) satisfies all persistency requirements.
3.3 Possible solution - Fowler’s Active Record

Applying the Active Record pattern as a straight forward solution would mean to overload each persistency marked class with all storage access methods (e.g. SELECT, UPDATE, DELETE). This solution doesn’t satisfy the solution requirements due to the following violations:

- The class marked as persistent (Message) has the added responsibility of managing its coexistence within the storage. Adding this new responsibility would mean to change the structure (and perceived notion) of this class in the domain layer (property (3)).

- Changing the persistent class structure also obligates changes to its clients. In an existing system, this would require to recompile all client classes (property (2)).

- Applying the Active Record pattern has another limit. The persistent object must stay memory resident to support all persistency services. Therefore, on idle times, the object cannot be released from memory (property (4)).

Conclusion: The Active-Record pattern fall short to satisfy the above solution requirements. Using the Data Mapper pattern seems to be a more promising solution. Keeping the persistent class intact and assigning all persistency handling aspects to a new independent class seems to suppress the bad symptoms of the Active Record solution.
3.4 Possible solution - Fowler’s Data Mapper

Based on Fowler’s *Data Mapper* pattern, a possible solution is to introduce a new data-mapper class that is responsible for persistency services of the class marked as persistent. A single object of the data-mapper class serves all the clients of the persistent class. The new data-mapper object stands between the persistent class and its client. For example, Figure 3.4 illustrates a *MessageDataMapper* class that has the role of a data-mapper class. For each client of the persistent class *Message* like *EditorSession* in the figure, the *MessageDataMapper* class implements all of its persistency services. The *MessageDataMapper* class may also be responsible for performance optimization of persistency services.

**Usage Example** Figure 3.5 demonstrates a possible course of events for executing the method *addAttachment()* on the client - html:*EditorSession* object. As illustrated in the figure, in order for the *EditorSession* client to communicate with an object of the persistent class, it must first address the mapper to load it. Since the request to the mapper must be accompanied with the id of the instance to be retrieved, the *EditorSession* must provide it. Therefore, the *EditorSession* has been added a new attribute *idOfMessage* to store this id. After changes had been made, the client should also address the mapper to store the message’s state back into the storage.
Solution Advantages  Applying Fowler's mapper in the context of the problem discussed has the following advantages:

1. Persistency considerations (e.g. SQL implementations) are handled by the mapper class and do not affect the class marked to be persistent.

2. Responsibility for the instantiation and destruction of objects of the class marked as persistent is imposed on the data-mapper.

Solution Drawbacks

1. **Added id field.** In case of a single mapper being shared by multiple client instances, an identity attribute must be added either to the persistent class or to the client class. Without the id field the mapper cannot identify the corresponding persistent instance in storage. Furthermore, this drawback violates the solution requirements since it implies changes to the
structure of domain classes (properties (2) and (3)).

2. **Handling outgoing references:** As illustrated in Figure 3.4, the persistent class may be also associated with other in-memory classes through an outgoing reference. For example, in a secured email system, the *Message* class may be associated with an *encryptionKey* class that represents keys for encrypting/decrypting the content of each message instance. Instances of the *encryptionKey* class are generated at runtime and there is no requirement to store them in storage. This kind of interrelationship with in-memory objects should be restored whenever loading a persistent class object. However, Fowler’s data mapper pattern lacks a solution for such kind of external links (property (4)).

3. **Maintaining duplications consistency:** Between the in-memory object representation and its storage one. When applying the datamapper pattern, there are two possible candidates to handle consistency:

   (a) **Imposed on the client.** *(EditorSession in the figure)* The client of the persistent class is responsible for retaining the in-memory instance consistent with the storage. This may be achieved using some sort of conflict resolution mechanism (e.g. the one used in a materialized view of an Oracle Database System [25]) or a delete-and-insert approach [9]. Disregarding to the mechanism selected, this kind of solution violates the second solution requirements impacting the Client class.

   (b) **Imposed on the persistent class itself.** *(Message in the figure)* Enriching the persistent class with the ability for the mirror in-memory copy of *Message* to alert its mapper of any changes done to it. This kind of solution entails the addition of navigability between the persistent class and its mapper. The result is actually very similar to using the *Active Record* pattern. Since in this case, the persistent class is given an extra responsibility to take care of synchronization with the DB.

4. **Wrong access attempts:** If duplications consistency handling is imposed on the persistent
class, the client object (e.g. EditorSession) may be oblivious to the existence of the persistent object in memory, wrong access attempts might occur. A possible solution for this drawback is to replace all unavailable objects of the persistent class with a single dummy facade object [11]. The dummy object role is to trigger the mapper to load the requested object from the storage back to memory. However such a solution mechanism should be tailored manually and is not the part of the datamapper solution.

5. Dealing with multiple clients: In Fowler’s datamapper pattern, there is no strict definition of how many datamapper objects may be used for each datamapper class. Therefore, in case where the persistent class Message has more than one client like EditorSession, there are two possible approaches:

(a) **Client oriented Mapper.** Each client has its own mapper object for accessing the persistent object. The disadvantage of this approach is the high possibility for duplicated objects created for the same database record. In this approach implementing a diff strategy is essential (e.g. Optimistic Offline Lock [9]) to resolve conflicts.

(b) **Persistency oriented Mapper.** A single Mapper object is created for each persistent class hence serving all its clients (e.g. implementing singleton pattern). This approach prevents consistency violation. There is no need to deal with conflicts and the class diagram is not overloaded with datamapper classes.

Fowler’s datamapper pattern doesn’t explicitly specify the nature of the mapper, whether it is client or persistency oriented. It seems that the latter provides a better solution.

**Conclusion:** Applying fowler’s datamapper pattern as a solution to persistency requirement has several disadvantages. This solution cannot be applied as straight forward since it doesn’t satisfy the solution requirements.
3.5 Improving the solution

3.5.1 1st Improvement - Hiding the mapper from the client

As implied from the conclusion, the existence of the mapper class should be hidden from the client of the persistent class. This may be achieved using simple inheritance as illustrated in figure 3.6. This mapper internally owns a message instance for which it supports all persistency services.

![Figure 3.6: A MessageDataMapper that inherits from Message](image)

The presented structural model satisfies all solution requirements except of the last one. The EditorSession client communicates with the MessageDataMapper class as if it was a simple Message. Therefore, the client is not aware of the existence of the mapper class. However, this solution implies that each mapper object has all the structure (state) of a message object, which is not the intention of a wrapper.

A significant difficulty: Each mapper object, except for holding its owned message instance, is chained to all of its super-typed message containment. This might result with in-memory inconsistencies between the mapper self state as a message and its attached message object reflected to the client (requirement (6)).

3.5.2 2nd Improvement - The Proxy Data Mapper

Following the GoF spirit [11], a better solution is achieved by combining both notions of the Proxy pattern and Fowler’s data mapper pattern. Our solution, called Proxy Data Mapper, is presented in
As illustrated in Figure 3.7 the solution architecture is mainly composed of:

1. An extracted interface (Message class in the figure): An abstract class that contains all public methods signatures extracted from the original persistent class with the addition of factory method(s).

2. An implementation class (MessageImp class in the figure): The original persistent class, with a modified name (postfix 'Imp').
3. A proxy class (MessageProxy class in the figure): Represents lightweight instances of the persistent class. Each instance of the proxy-class contains an ID attribute for identifying the persistent instance in-storage and a set of delegation methods corresponding to all public methods of the persistent class.

4. A datamapper class (MessageDataMapper class in the figure): As in Fowler’s datamapper, the class is responsible for wrapping all DB services (SELECT, INSERT, UPDATE and DELETE) and for the local construction of the full-weight representative of the persistent class (MessageImp in the figure).

The solution main idea is replacing the in-memory fullweight instance of the persistent class with only 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., EditorSession) 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.

Only at runtime, when any service of the persistent class instance is required, the proxy instance notifies the datamapper class object to re-construct a fullweight in-memory duplication of the persistent class and invokes the corresponding method implementation of the service required by the client. After service activity has terminated, the proxy instance saves the persistent instance state back to storage (through the datamapper) and releases the duplication from being memory resident. Synchronization between in-memory and persistent objects is imposed on the new MessageProxy.

In-memory duplications cannot occur because the datamapper object uses an identity map to ensure that each persistent object is loaded to memory at most once. Therefore, for each persistent class there is also a corresponding identity map accessible to the datamapper.
The hiding of the full persistent object behind their proxies is actually carried out by a *factory method* that must replace all constructors of the persistent class. The *factory method* returns a proxy object (e.g., *MessageProxy*). This way clients of the persistent class are not affected by this pattern.

### 3.5.3 Proxy Data Mapper Usage Example

Figure 3.8 describes a request of the in-memory client object `html:EditorSession` to apply `set-Subject(sub)` on its associated persistent object `mes:MessageImp` (which the client recognizes as a *Message* object). *EditorSession* access to `mes:MessageImp` is done through the *proxy* class object `mes:MessageProxy`. A request to the *proxy* to execute a method on its encapsulated persistent object triggers three actions:

1. **Load** the persistent object from storage to memory. This operation is handled by the *datamapper* object (*mapper:MessageDataMapper*).

2. **Forward** the requested operation to the loaded object.

3. **Update** its state within the storage. This operation is handled by the *datamapper* object (*mapper:MessageDataMapper*).

The *EditorSession* client is oblivious to whether messages are persistent or not. Therefore, both the client and the persistent class are not changed by the pattern application.

Corresponding to all problems detailed above, the application of the *Proxy-Data-Mapper* pattern imply satisfaction of *solution requirements* as follows:

1. All instances of a persistent class are stored within storage as handled by the responsibility of the new datamapper class.

2. All clients are oblivious to persistency handling - achieved by the preservation of the original interface of the class that is marked persistent and instance construction that is achieved by the invocation of factory method.
3. Domain-Layer classes stay intact. There is only a minor change to the name of the persistent class that can be easily traced given the input of persistency marking.

4. Instances of the persistent class are being loaded to their fullweight representation and released whenever needed by clients as handled by their lightweight representative (proxy). **However, the presented solution is only local for each persistent class**: The illustrated solution doesn’t deal with restoring outgoing references to other in-memory or persistent objects.

5. Consistency maintenance between in-memory state and storage state of persistent instances is imposed on the proxy class.

6. Avoiding in-memory duplications is handled by the implementation of factory methods. Since
proxys function as thin in-memory representatives of stored objects, the solution is to avoid duplicated copies of proxys. Therefore, the factory method implements Fowler’s "Identity map" pattern [9] to ensure each proxy instance is loaded only once using a Look-Up map of objects that are already loaded.

**Conclusion:** the **Proxy-Data-Mapper** pattern satisfies solution requirements only within a local context of the class marked as persistent, thereby perceived as the **core pattern**. Expanding the solution scope to deal with outgoing references is to be achieved by a set of context patterns presented in chapter 4.

### 3.6 Proxy Data Mapper - Detailed Description

#### 3.6.1 Proxy Data Mapper - Input

1. **persistentClass**: A class object.

2. **dbSchemaElement**: mapping definition between the persistent class and some storage element (i.e. a Table). An example for a possible dbSchemaElement correlated with the class Message is illustrated in Figure 3.9.

```xml
<mapping-class-to-dbschemaElement>
  <class name="myApplication.Message" table="Messages">
    <attribute name="subject" field="title"/>
    <attribute name="body" field="body"/>
  </class>
</mapping-class-to-dbschemaElement>
```

Figure 3.9: Example for dbSchemaElement

#### 3.6.2 Proxy Data Mapper - pre-conditions

1. **persistentClass** is marked as persistent.

2. **persistentClass** has factory methods that replace every necessary constructor. The factory methods apply constructors based on external information.
3.6.3 Proxy Data Mapper - post-conditions

With respect to the **persistent-class**:

1. An *abstract-class* with the same name of the *persistent-class* exists.

2. The *persistent-class* inherits from the *abstract-class* and the name of the class is extended with the postfix "Imp".

3. A *datamapper* class exists and bridges between the storage and the *persistent-class*.

4. A *proxy* class exists and inherits from the *abstract-class*.

5. The *proxy* class encapsulates the *persistent-class* and delegates to it.

6. The *abstract-class* contains a *factory-method* to construct instances of the *proxy* class linked to the *datamapper* class.

3.6.4 Proxy Data Mapper - Action

The application of the *Proxy Data Mapper* pattern is a composed sequential application of *Transformation Building Blocks* described in Chapter 8. We provide here a concise description:

1. **Extract the implementation from the persistentClass:**
   
   ```
   persistentClassImp = extractSubClass(persistentClass);
   ```

2. **Construct the data mapper class:**
   
   (a) Add a new class:
   
   ```
   dataMapper = addClass(concat(persistentClass.name+'''DataMapper'''));
   ```

   (b) Add a *load* method to the new DataMapper class. This action uses the *loadSignature(...)* transformation that returns a signature for a load method typed as the persistentClass and *loadImp(...)* transformation that return an implementation for the *load* method with all
embedded data access (i.e. SQL statements) to retrieve and construct a new instance of
the persistent class from storage given the corresponding dbSchemaElement.

```java
addMethod(dataMapper, loadSignature(persistentClass),
           loadImp(persistentClass, dbSchemaElement));
```

(c) Add a *store* method to the new DataMapper class. This action uses the `storeSignature(...)` transformation that returns a signature for a store method with an argument typed as the persistentClass and `storeImp(...)` transformation that return an implementation for the *store* method with all embedded data access (i.e. SQL statements) to store all details of the given instance of the persistent class into storage given the corresponding dbSchemaElement.

```java
addMethod(dataMapper, storeSignature(persistentClass),
           storeImp(persistentClass, dbSchemaElement));
```

(d) Add an *update* method to the new DataMapper class. This action uses the `updateSignature(...)` transformation that returns a signature for an update method with an argument typed as the persistentClass and `updateImp(...)` transformation that return an implementation for the *update* method with all embedded data access (i.e. SQL statements) to update all details of the given instance of the persistent class into storage given the corresponding dbSchemaElement.

```java
addMethod(dataMapper, updateSignature(persistentClass),
           updateImp(persistentClass, dbSchemaElement));
```

(e) Add a *delete* method to the new DataMapper class. This action uses the `deleteSignature(...)` transformation that returns a signature for a delete method with an id argument and `deleteImp(...)` transformation that return an implementation for the *delete* method with all embedded data access (i.e. SQL statements) to delete the given instance id from storage given the corresponding dbSchemaElement.
addMethod(dataMapper, deleteSignature(persistentClass),
deleteImp(persistentClass, dbSchemaElement));

**Note:** The resulting DataMapper class is exactly like Fowler’s datamapper: Assumes only
Data-Layer ↔ Database connecting responsibilities. It is detached of the persistent class.

3. **Construct the proxy class:**

(a) Add a new class:

```java
proxy = addClass(concat(persistentClass.name+''Proxy''));
```

(b) Add two private fields to the proxy class: an internal private ID field and an internal
private reference typed as a DataMapper class.

```java
addField(proxy,_idField);
addField(proxy,_mapperField);
```

(c) Add three private methods to the proxy: finalize method that delegates to the delete
method of the DataMapper, load method that uses the load service of the DataMapper
and update the uses the update service of the DataMapper. All method implementa-
tions are generated using the InvokeDelete(...), InvokeLoad(...) and InvokeUpdate(...)
transformations respectively.

```java
addMethod(proxy,finalizeSig(),InvokeDelete(dataMapper));
addMethod(proxy,loadSig(dataMapper),InvokeLoad(dataMapper));
addMethod(proxy,updateSig(dataMapper),InvokeUpdate(dataMapper));
```

(d) Add a corresponding constructor to the proxy for each constructor implementation of
the concrete persistentClassImp. This action uses the signatureOf(...) transformation to
extract the signature from each constructor of the persistentClassImp. The implementa-
tion of each constructor is added using the InvokeStore(...) transformation that generates
a new concrete persistentClassImp instance and uses the store method of the mapper to
store the new instance into storage.
for each constructor c in constructors(persistentClassImp)
    addConstructor(proxy, signatureOf(c),
        InvokeStore(dataMapper,persistentClassImp));

(e) Redirect all factory methods of the persistentClass to return an instance of the new generated proxy:

    for each factoryMethod f in factories(persistentClass)
        redirectFactoryMethod(proxy);

(f) For each instance method of the concrete persistentClassImp, add a corresponding delegated method with the same signature:

    for each method m in instance_methods(persistentClassImp)
        addDelegatedMethod(proxy, signatureOf(m),
            persistentClassImp);

4. **Tag modifications**:

   Change the persistentClass to be an abstract class:

   stereotype(persistentClass, "abstract");

For the usage of the following context patterns, the application of the Proxy-Data-Mapper pattern also stores **meta-data** information with the names of the proxy and mapper classes that were generated. The code for a *MessageProxy* class, generated for the *Message* class by applying the described transformations is given in listing 3.1.

By applying the **Proxy Data Mapper**, the client class is oblivious to the fact that all of its requests are dealt by a proxy's instance. The client communicates with the proxy through the interface extracted from the persistent class. All **local** persistency requirements are handled by a datamapper through the proxy object. And most important, no changes were required to be applied neither on the client nor on the persistent class. Therefore, the **Proxy Data Mapper** pattern satisfies **solution**
requirements in the local context of the persistent class. However, since in real-world complex systems, the existence of the persistent class also involves various incoming and outgoing relations with other classes (not necessarily persistent). A global solution for persistency requirements must be satisfied. Such a solution is achieved by the application of context specific patterns each to deal with a different context of the class marked as persistent. All context patterns are presented in the following chapter.
Listing 3.1: MessageProxy class

```java
public class MessageProxy extends Message {

    private int ID;
    private MessageDataMapper dataMapper;

    public MessageProxy(String title) {
        dataMapper = MessageDataMapper.getUniqueInstance();
        ID = dataMapper.store(new MessageImp(title));
    }

    protected void finalize() throws Throwable {
        dataMapper.delete(ID);
    }

    private MessageImp load() {
        return dataMapper.load(ID);
    }

    private void update(MessageImp mes) {
        dataMapper.update(mes, ID);
    }

    // Delegated methods:

    public void addAttachment() {
        MessageImp mes = load();
        mes.addAttachment();
        update(mes);
    }

    public String getBody() {
        Message mes = load();
        String result = mes.getBody();
        update(mes);
        return result;
    }

    ... more delegated methods
}
```
Chapter 4

Data access patterns catalog

The core pattern is used for wrapping all persistently marked classes with persistency services. The comprehensive solution to persistency requirements is achieved by applying the Context Data Access Pattern corresponding to the context of each class marked as persistent in the system. The context of the persistently marked class is determined by a combination of the following three parameters:

1. **Navigation.** The direction of the association connecting the persistent class and the context class. The navigation may be either an incoming association (from the context class to the persistent one) or an outgoing association (from the persistent class to the context one).

2. **Cardinality.** This parameter makes a distinction between an association to a single object (marked with "1") and an association to multiple objects (marked with "*").

3. **Context class.** The context class may be stereotyped as being either an in-memory class or a persistent one.

Combinations of these three parameters give rise to five different context data access patterns. All patterns are listed in table 4.1.

A context pattern may be conceived as a condition-action rule composed of:

The context singles out the persistent class context in the domain layer to which the pattern applies. The pre-condition singles out a constraint that must hold prior to the execution of the action part of a pattern. The action describes the suggested transformation and the post-condition describes a constraint that must hold after the pattern is applied.

Sections 4.1 to 4.5 are a detailed catalog of all Context Data Access Patterns.

**Figures legend:** The presentation of each pattern is accompanied by some class-diagram figures. All figures single out the following notations:

- - - - A dashed line that connects between two classes:
  Represents temporal visibility between objects. For example: an object of one class is responsible for only the instantiation of the other. In such case, the classes of these objects are associated with a dashed line.

——— A flat line that connects between two classes:
  Represents permanent visibility between objects. For example: an object of one class has a permanent reference to another object. In such case, the classes of these objects are associated with a flat line.

<table>
<thead>
<tr>
<th>Context</th>
<th>Pattern name</th>
</tr>
</thead>
<tbody>
<tr>
<td>context-class(in-memory) → persistent-class</td>
<td>Persistent Collection Data Mapper</td>
</tr>
<tr>
<td>context-class(in-memory) ← 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) → persistent-class</td>
<td>Persistent to Persistent</td>
</tr>
<tr>
<td>context-class(persistent) → persistent-class</td>
<td>Persistent to Multi Persistent</td>
</tr>
</tbody>
</table>

Table 4.1: Context Data Access Patterns
• A “lock-icon” that appears next to an attribute or a method name:

    Represents a *private* access level to the corresponding attribute or method.

All other notations are the same as in UML class-diagram notation.
4.1 Persistent Collection Data Mapper

Context  An in-memory collection of multiple instances of a class marked as persistent.

The context of the Persistent Collection Data Mapper pattern is identified by an in-memory collection (Collection class in figure 4.1) responsible for handling multiple instances of a class that is marked as persistent (Message class in figure 4.1).

For example, in an email application, the context collection may be the realization of the conceptual association between the VirusScan class and instances of class Message associated to it as its scanning targets as illustrated in figure 4.2.

Therefore, the pattern applies for the following combination of the three parameters of the context patterns:
Problems that arise in the context of the *Persistent Collection Data Mapper* pattern:

1. **The problem of in-memory collection realization:** Applying only the core pattern (PDM) as the pre-condition for this pattern application does achieve local persistency handling to the persistent class. Each stored persistent object, while being non-active is replaced with its in-memory lightweight proxy representative. Therefore, the context collection actually manages a set of proxy objects instead of fullweight in-memory objects. Such solution may satisfy small collection sets. However, when having larger collection sets of persistent objects, the in-memory residence of proxy objects, each with its small containment of data (ID) and delegated method implementations, might aggregate to be resource consuming.

The in-memory existence of the collection should be reduced to minimum. Unless the lightweight proxy instance is *essential* (see section 4.1.2) it should exist only for active elements whenever they are manipulated by clients of the collection. During their idle periods, each collection member may be represented by only a minimal footprint (possibly its ID) within the collection. The corresponding proxy instance may be released from being memory resident. However, releasing the proxy instance exposes the *deletion problem* as explained bellow.

2. **The deletion problem:** Loosing the reference to the proxy instance for each element that is represented by a minimal footprint as implied from the first problem, might initiate the deletion process of the real persistent object from storage. However, for elements in the collection that are represented only by their footprint this reaction should be deprecated. An element that is represented by its small footprint should still be considered as included within the collection.

In a simple setting, where all object references reside in memory alone, deletion of an in-memory object which is a copy of a stored object should imply deletion of the real object.
from storage (as discussed in the context of the Proxy Data Mapper pattern). However, when object references reside in “footprints” in memory collections, this is not the case anymore, since it is possible that although all in-memory references to the lightweight representative are lost, there are still footprint references.

Problem solutions:

1. The solution to collection realization problem: The structure of the suggested solution is illustrated in figure 4.3. It involves the addition of a context-specific collection class \( \text{MessagesCollectionMapper} \). The added collection is responsible for loading and unloading proxy objects to memory per client request for each element included in the collection. 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. \text{List}). The construction of the new collection is achieved by the factory method of the collection being redirected.

2. Solution to the deletion problem: To solve this problem, the proxy class is associated with a new Data Access Layer class named \text{TableManager}. Since each persistent instance may be referenced either from an in-memory client or from the context-specific collection, the \text{TableManager} class has the responsibility for guarding storage deletion of each persistent instance by tracking reference count to it. The \text{TableManager} propagates a deletion call to the mapper of the persistent object only when no more references are pointing to it. That is, as long as there is an in-memory proxy for a persistent instance, or a memory collection that holds its id, the persistent instance is kept in the persistent layer.

To prevent inconsistencies with storage state and to assure a single entrance point for each storage table, a single instance of the \text{TableManager} serves all of its clients (implemented as a singleton).

The insertion of a \text{TableManager} in between the memory objects and the stored ones mean that a proxy that has an associated \text{TableManager} has to delegate storage deletion calls to
its TableManager instead of delegating it to its datamapper (as in the Proxy-Data-Mapper pattern in section 3.5.2). In addition, the context-specific collection notifies the TableManager class about containment changes (additions and removals) as well. For example, the MessagesTableManager in Figure 4.3 allows storage deletion of a concrete message object, only when this message object is not contained within any MessagesCollectionMapper and no more instances of the MessageProxy class are referring to it.

The full action of this pattern that solves the above problems is described bellow.

4.1.1 Persistent Collection Data Mapper - Detailed Description

4.1.1.1 Persistent Collection Data Mapper - Input

The application of this pattern requires the following inputs:

1. collection : A collection class of persistent objects.
2. persistentClass : The persistent class of these objects.

4.1.1.2 Persistent Collection Data Mapper - Pre-conditions

1. The post-condition of the Proxy-Data-Mapper pattern holds with respect to the class that is marked a persistent.

4.1.1.3 Persistent Collection Data Mapper - Post-condition

1. A context-specific-collection class exists with the name of the persistentClass extended with the postfix "sCollectionMapper".
2. The context-specific-collection class inherits from a regular library collection class (e.g. List).
3. A tableManager class exists to handle destruction of persistentClass instances.
4. Both the context-collection and the proxy classes are associated with the tableManager class.
4.1.1.4 Persistent Collection Data Mapper - Action

The action of the *Persistent Collection Data Mapper* pattern is composed of sequential application of the following Transformation Building Blocks described in chapter 8:

1. **Create TableManager (problem 2):**
   
   If does not exist already, create a new singleTon tableManager for the persistent class. The creation also associates the new tableManager with the mapper of the persistentClass (the name of the mapper class is resolved from the meta-data information that is generated by the Proxy Data Mapper):

   ```java
   if (existTableManager(persistentClass) == false)
   then
     createTableManager(persistentClass);
   ```

2. **Assign Proxy (problem 2):**

   If not yet assigned, assign the proxy class to notify the table manager on construction and destruction of proxy instances:

   ```java
   if (isProxyAssigned(persistentClass.proxy,
                        persistentClass.tableManager) == false)
   then
     assignTableManager(persistentClass.proxy, persistentClass.tableManager);
   ```

3. **Create Collection (problem 1):**

   Create the context specific collection and assign it to the proxy and to the tableManager of the persistent class (the name of the proxy class is resolved from the meta-data information that is generated by the Proxy Data Mapper). The creation of the new collection also redirects the factory method of the in-memory *collection* class to the new collection:

   ```java
   createPersistentCollectionClass(collection,
   ```
4.1. Persistent Collection Data Mapper

```
persistentClass.proxy,
persistentClass.tableManager);
```

The application result is illustrated in Figure 4.3. The additions of the Persistent Collection Data Mapper pattern application is singled out with classes colored in gray background.

**Extended Action:** Sometimes we wish to make the collection itself persistent. This may be because the collection is too big. Another reason may be the wish to preserve it permanently, independently of whether there is an in-memory object that holds it. To preserve the existence of a collection instance, we suggest to apply the Proxy-Data-Mapper pattern to the collection itself. The final result is illustrated in Figure 4.4. The additions of the Proxy Data Mapper pattern application to the collection is singled out with classes colored in gray background.

4.1.2 Persistent Collection Data Mapper - Related Patterns

- Memoization patterns:
  - *Memoization Proxy Data Mapper* pattern
  - *Multi Memoization Proxy Data Mapper* pattern

The application of both memoization patterns makes the proxy class an essential class. Therefore, if either one of the two related patterns was applied on the persistent class in the collection, the persistent collection class cannot replace the lightweight representative with a footprint as suggested in this pattern. In such case, the collection remains a collection of proxy objects.

- *Persistent to Multi Persistent* pattern: This pattern must be applied after the application of the persistent-to-multi-persistent pattern since it is indifferent regarding the context of the persistent collection (i.e. if the owner of the collection is persistent or not). Therefore, if the collection appears within the context of the persistent-to-multi-persistent pattern, the application of the later should be tried first.
4.1. Persistent Collection Data Mapper

Figure 4.3: Structure introduced by the *Persistent Collection Data Mapper*

- **Proxy Data Mapper pattern**: must be applied before the application of this pattern.
Figure 4.4: Persistent Collection Data Mapper - Extended Action Result
4.2 Memoization Proxy Data Mapper

**Context** An instance of a persistent class has to maintain its connection to an instance of an in-memory class.

![Figure 4.5: Memoization Proxy Data Mapper - Context](image)

Sometimes, the persistent class is associated with a context class that represents instances that are meant to reside only in-memory. Such restrictions can arise in different scenarios. For example, in an email application, each message may be encrypted using an encryption-key (as illustrated in Figure 4.5). The encryption of each message is executed before each message is being sent. However, the encryption-key is relevant only during a single interaction between a sender and its addressee. Therefore the existence of each encryption-key object is temporal and due to security requirements is also forbidden from being kept within any storage. This pattern applies for the case where a single instance of a persistent class is referencing a single instance of an in-memory class. Therefore, the *Memoization Proxy Data Mapper* pattern applies for the following combination of the three parameters of the context patterns:

<table>
<thead>
<tr>
<th>Navigation</th>
<th>Cardinality</th>
<th>Context class</th>
</tr>
</thead>
<tbody>
<tr>
<td>outgoing</td>
<td>1</td>
<td>in-memory</td>
</tr>
</tbody>
</table>

**The Memoization Problem** As mentioned in section 3.4, the application of the core *Proxy-Data-Mapper* pattern corresponding to the class that is marked as persistent, does not handle outgoing references from that class. Having only the core pattern applied in the context of this pattern implies
that whenever an instance of a persistent class is being represented by its lightweight in-memory proxy, all outgoing references to other linked objects are neglected. Therefore, whenever a persistent instance is being loaded from storage to its fullweight in-memory residence, there is no mean for restoring it to its previous linkage state.

The solution for the Memoization Problem  The solution involves extending each proxy class with a reference attribute for each outgoing relation from the original persistent class. This reference preserves the outgoing link and existence of each referred object, when their persistent owner is stored. Whenever the persistent object is reloaded to its fullweight state, the proxy restores all linked objects to it. Each reference state is updated whenever the persistent object is being stored.

4.2.1 Memoization Proxy Data Mapper - Detailed Description

4.2.1.1 Memoization Proxy Data Mapper - Input

The application of this pattern requires the following inputs:

1. persistentClass: A class object.

2. referredClass: An outgoing reference to a class typed as a regular in-memory class (reference attribute).

4.2.1.2 Memoization Proxy Data Mapper - Pre-condition

1. With respect to the persistent-class, the post-condition of the Proxy-Data-Mapper core pattern holds.

4.2.1.3 Memoization Proxy Data Mapper - Post-Condition

1. The persistent-proxy class has a reference attribute to the referredClass.
4.2.1.4 Memoization Proxy Data Mapper - Action

The action of the Memoization Proxy Data Mapper pattern is composed of sequential application of the following Transformation Building Blocks described in chapter 8:

1. **Add reference attribute to the Proxy class:**

   Revise the proxy of the persistent class to include a reference attribute typed with the same type of the outgoing referred class:

   \[
   \text{addReferenceAttribute(persistentClass.proxy,referredClass)}
   \]

2. **Revise Proxy Methods:**

   (a) **Revise all constructor methods:**

   All constructor methods of the proxy class are revised to update the value of the added reference attribute after constructing a new persistent-class object:

   \[
   \text{for each constructor in constructors(persistentClass.proxy)}
   \text{updateReferenceValueInConstructor(constructor,referredClass)}
   \]

   (b) **Revise the private update method:**

   The *update* method of the proxy class is revised to update the value of the added reference attribute just before updating it in storage (via the datamapper):

   \[
   \text{updateMethod = getUpdateMethod(persistentClass.proxy)}
   \text{updateReferenceValueInUpdateMethod(updateMethod,referredClass)}
   \]

   (c) **Revise the private load method:**

   The *load* method of the proxy class is revised to re-attach the referred object (value of the reference attribute) to the loaded fullweight representative of the persistent class:

   \[
   \text{loadMethod = getLoadMethod(persistentClass.proxy)}
   \text{reAttachReferenceInLoadMethod(loadMethod,referredClass)}
   \]
4.2. Memoization Proxy Data Mapper

The Memoization Proxy Data Mapper pattern application result is illustrated in Figure 4.6. The code for a revised MessageProxy class, generated for the Message class by applying the described transformations is given in listing 4.1

4.2.2 Memoization Proxy Data Mapper - Related Patterns

- **Persistent Collection pattern** The application of this pattern entails that each proxy object that has a memoization responsibility cannot be deleted unless it is replaced with another object that takes that responsibility. Therefore, if this pattern was applied on a class that is also a member of a persistent collection (section 4.1), the implementation of the collection cannot allow deletion of lightweight representatives.

- **Proxy Data Mapper pattern**: must be applied before the application of this pattern.
Figure 4.6: Structure introduced by the Memoization Proxy Data Mapper
4.3 Multi Memoization Proxy Data Mapper

**Context**  
An instance of a persistent class has to maintain its connection to a collection of in-memory objects.

Similar to the *Memoization-Proxy-Data-Mapper* pattern, each persistent instance may be associated with a collection of in-memory objects that are either forbidden from being stored in storage or instances that their existence is only temporal within some process. For example, in an email application, all previous states of a message that is being edited are recorded to enable rollback (undo) to one of its previous states. Furthermore, the rollback data is relevant to a message only while it is being edited. Once the message has been sent, the rollback data of a message is dumped. Therefore, each message object is linked to a collection of *UndoData* objects (illustrated in Figure 4.7). This pattern applies for the case where a single instance of a persistent class (e.g. *Message* in Figure 4.7) is related to multiple instances of an in-memory class (e.g. *UndoData* in Figure 4.7). Therefore, the *Multi-Memoization-Proxy-Data-Mapper* pattern applies for the following combination of the three parameters of the context patterns:

<table>
<thead>
<tr>
<th>Navigation</th>
<th>Cardinality</th>
<th>Context class</th>
</tr>
</thead>
<tbody>
<tr>
<td>outgoing</td>
<td>*</td>
<td>in-memory</td>
</tr>
</tbody>
</table>

![Diagram](image-url)  
*Figure 4.7: Multi-Memoization Proxy-Data-Mapper - Context*
The Multi-Memoization Problem  The multi-memoization problem is the same problem as the memoization problem described in the Memoization-Proxy-Data-Mapper pattern, but with respect to a collection of in-memory objects instead of just a single instance. The Proxy-Data-Mapper pattern by itself does not support the ability to restore any of the outgoing links to in-memory instances (or collections of in-memory instances in this case) when a persistent instance is being reloaded to its in-memory fullweight representation.

The solution for the Multi-Memoization Problem  Since the problem is almost identical to the Memoization Problem described in the Memoization-Proxy-Data-Mapper pattern, the solution is almost the same. It involves extending each proxy class with a reference attribute for each outgoing relation from the original persistent class. The only difference is that in the Multi-Memoization solution, the added reference attribute is typed as collection. The role of this reference is the same as in the memoization pattern, that means to preserve the outgoing link and existence of each referred collection of objects, when their persistent owner is stored. Whenever the persistent object is reloaded to its fullweight state, the proxy restores the linked collection to it. Each reference state is updated whenever the persistent object is being stored.

4.3.1 Multi Memoization Proxy Data Mapper - Detailed Description

4.3.1.1 Multi Memoization Proxy Data Mapper - Input

The application of this pattern requires the following inputs:

1. persistentClass : A class object.

2. collectionClass : An outgoing reference to a class typed as a collection class (i.e. List).

4.3.1.2 Multi Memoization Proxy Data Mapper - Pre-condition

With respect to the persistent-class, the post-condition of the Proxy-Data-Mapper core pattern holds.
4.3.1.3 Multi Memoization Proxy Data Mapper - Post-condition

1. The persistent-proxy class has a reference attribute to a collection-class.

2. The methods load(), update() and constructor in the persistent-proxy class handle consistency of the collection-class with its appearance in storage.

4.3.1.4 Multi Memoization Proxy Data Mapper - Action

The action of the Multi Memoization Proxy Data Mapper pattern is composed of sequential application of the following Transformation Building blocks described in chapter 8.

1. **Add reference attribute to the Proxy class:**

   Revise the proxy of the persistent class to include a reference attribute typed as the collectionClass:

   \[
   \text{addReferenceAttribute(persistentClass.proxy, collectionClass)}
   \]

2. **Revise Proxy Methods:**

   (a) **Revise all constructor methods:**

      All constructor methods of the proxy class are revised to update the value of the added reference attribute after constructing a new persistent class object:

      \[
      \text{for each constructor in constructors(persistentClass.proxy)}
      \]

      \[
      \text{updateReferenceValueInConstructor(constructor, collectionClass)}
      \]

   (b) **Revise the private update method:**

      The update method of the proxy class is revised to update the value of the added reference attribute just before updating it in storage (via the datamapper):

      \[
      \text{updateMethod = getUpdateMethod(persistentClass.proxy)}
      \]

      \[
      \text{updateReferenceValueInUpdateMethod(updateMethod, collectionClass)}
      \]
4.3. Multi Memoization Proxy Data Mapper

(c) Revise the private load method:

The load method of the proxy class is revised to re-attach the referred collection (value of the reference attribute) to the loaded fullweight representative of the persistent class:

```java
loadMethod = getLoadMethod(persistentClass.proxy)
reAttachReferenceInLoadMethod(loadMethod, collectionClass)
```

The Multi Memoization Proxy Data Mapper pattern application result is illustrated in Figure 4.8.

4.3.2 Multi Memoization Proxy Data Mapper - Related Patterns

- **Persistent Collection pattern** The application of this pattern entails that each proxy object that has a collection-memoization responsibility cannot be deleted unless it is replaced with another object that takes that responsibility. Therefore, if this pattern was applied on a class that is also a member of a persistent collection (section 4.1), the implementation of the collection cannot allow deletion of lightweight representatives.

- **Proxy Data Mapper pattern**: must be applied before the application of this pattern.
4.4 Persistent to Persistent

**Context** An instance of a persistent class is a client of an instance of another persistent class.

This pattern applies to cases where a single instance of a persistent class references a single instance of another persistent class, using a permanent reference, i.e., using a reference attribute. For example, in the e-mail client application example, each Message object may contain an attached
file that is expressed as an instance of the class Attachment as illustrated in Figure 4.9.

The context of the Persistent-to-Persistent pattern is identified by the following combination:

<table>
<thead>
<tr>
<th>Navigation</th>
<th>Cardinality</th>
<th>Context class</th>
</tr>
</thead>
<tbody>
<tr>
<td>outgoing</td>
<td>1</td>
<td>persistent</td>
</tr>
</tbody>
</table>

Problems that arise in the context of the Persistent-to-Persistent pattern: The context of the Persistent-to-Persistent pattern leads to problems that are caused by the need to manage duplicated in-memory and in-storage references. They arise when a persistent object, e.g., a message, that keeps a permanent reference to another context persistent object, e.g., an attachment, is loaded into memory, or is updated in memory, or is deleted from memory, or is created (stored).

1. **The load problem:** Loading the persistent object must realize also the permanent reference to its context object. The problem is what sort of realization the context loading should take: Should the context object be fully loaded, or should just a thin proxy representative be loaded. In any case, since both objects are loaded in either a full or a lightweight format, the in-storage reference is duplicated. For example, an in-memory copy of a Message instance is generated when some Message service, e.g., addSignature(), is invoked. The message instance must be associated with its attachment instance, implying that the outgoing reference must be realized in memory.

2. **The update problem:** The in-memory copy of the persistent object duplicates the in-storage reference to the context object. Changes made to the in-memory copy and its context must be recorded in storage, including cases of changed context, e.g., when the attachment is replaced.

3. **The deletion problem:** The same problem as described in the Persistent Collection Data Mapper pattern arises here. When a full in-memory copy of the persistent object is deleted from memory, e.g., when its services are completed, its in-memory reference to the context object is lost. However, both the persistent and the context objects still reside in storage. If there are no other in-memory references to the context object, it is possible that it is also deleted from memory.
For example, when an in-memory copy of a stored message instance is destructed, while its lightweight representative still lives in memory, its reference to the associated attachment instance (which might be a full copy or just a lightweight representative) is lost. This might lead to in-memory destruction of the context attachment instance. However, it should not imply deletion of the in-storage attachment object, since the in-storage reference is still there.

4. **The creation problem:** When the persistent object is created, its context is also created (it is a permanent visibility context). Both objects, being marked as persistent should be stored, and removed from memory, in case they were explicitly created as in-memory objects. The reference must be stored as well; otherwise, it will be lost.

**Problem solutions:**

1. **Solution to the load problem:** The responsibility to realize the reference to the context object is imposed on the datamapper of the persistent object. When loading a persistent object to memory, its datamapper also reconstructs a thin representative for the context object and re-attaches it as illustrated in Figure 4.10. This solution relies on the two assumptions:

   (a) **Existence of in-storage reference:** The reference between the persistent and context objects is indeed kept in the Data-Layer storage. Furthermore, the mapper can resolve the identity of the context object when loading the persistent object. For example, given a `foreighKey` column name as the reference identifier in storage, the mapper may resolve its value as the identifier of the context object (which is exactly the same value that identifies the context object in a Relational storage implementation).

   (b) **Accessibility to the persistent object:** The in-memory reference field that associates the persistent object to its context object must be accessible to the mapper. Otherwise, the mapper cannot set its value in-order to restore the state of the persistent object when loading it from storage. In such cases, Java Reflection may be used as an alternative.

Since both representatives reside in the *Data Access Layer*, the solution does not affect the
2. **Solution to the update problem:** All local changes to the in-memory copies (persistent and context objects) are handled by their corresponding datamappers. Changes in reference to the context object are handled by the datamapper of the persistent object. This is achieved by revising the `store()` and `update()` services of the datamapper of the persistent object, so that they keep track of changes in reference to the context object and update storage accordingly. Both assumptions of the load problem solution are valid in this solution as well.

3. **Solution to the deletion problem:** The solution is identical to the one described in the *Persistent Collection Data Mapper* pattern. The proxy of the context class is associated with a *TableManager* and revised to delegate the storage deletion procedure to its *TableManager*. In addition, the persistent datamapper notifies the `tableManager` about storage reference changes as well. For example, the datamapper of *Message* should notify the *TableManager* of *AttachmentDataMapper* whenever an attachment is added or removed from its message object.

4. **Solution to the creation problem:** Same as for the update problem.
4.4. Persistent to Persistent

4.4.1 Persistent to Persistent - Detailed Description

4.4.1.1 Persistent to Persistent - Input

Since the persistency situation of both classes is symmetric we can think of either class as *the persistent* class, while the other one plays the role of the context class. Since the major load of solving the problems seem to fall on the class that holds the reference, we refer to the latter as *the persistent* class, and to the referenced class as *the context* class. In Figure 4.9, *Message* is *the persistent* class, while *Attachment* is *the context* class. Indeed, this pattern can be viewed as a persistent version of the *Memoization Proxy Data Mapper* pattern. Nevertheless, since both classes are persistent the problems and the solutions are quite different.

Clearly, in a bi-directional reference situation a class can take both roles. Also, the same context class instance may be shared by multiple references that exist either in-memory or in-storage. Therefore, the application of this pattern requires the following inputs:

1. A persistent class referred to as *the persistent* class.

2. A persistent class referred to as *the context* class

4.4.1.2 Persistent to Persistent - Pre-condition

1. The post-condition of the *Proxy-Data-Mapper* holds with respect to both classes *the context* and *the persistent*.

2. Data layer implementation of the persistent class includes an attribute entitled *foreignKey* that provides an in-storage account for the reference from a persistent object to the related context object. An alternate name may be defined in the Domain-Data mapping specification that is given as an input for the application.
4.4.1.3 Persistent to Persistent - Post-Condition

1. A table-manager class for the context class exists to guard instance storage deletion of the context class.

2. Both the context proxy and the persistent datamapper classes are associated with the table-manager class of the context class.

4.4.1.4 Persistent to Persistent - Pattern Action

The action of the pattern is composed of sequential application of the following Transformation Building Blocks described in chapter 8.

1. **Introduce TableManager (problem 3):**
   
   Introduce a new tableManager to the contextClass. This transformation is the same as the first two transformations in the Persistent Collection Proxy Data Mapper pattern action detailed in section 4.1.1.4:

   ```
   IntroduceTableManager(contextClass)
   ```

2. **Assign the datamapper to the TableManager (problem 3):**
   
   Set the datamapper of the persistentClass to notify the tableManager of the context class about changes in references to instances of the contextClass:

   ```
   notifyToTableManagerOnDelete(persistentClass.mapper, tableManager)
   notifyToTableManagerOnStore(persistentClass.mapper, tableManager)
   notifyToTableManagerOnUpdate(persistentClass.mapper, tableManager)
   ```
3. **Revise the datamapper to handle the reference duplication (problems 1,2,4):**

Set the datamapper of the persistentClass to update reference change in storage and to realize it on load:

```java
updateStorageRefInStoreMethod(persistentClass.mapper, contextClass)
updateStorageRefInUpdateMethod(persistentClass.mapper, contextClass)
restoreRefInLoadMethod(persistentClass.mapper, contextClass)
```

The *Persistent to Persistent* pattern application result is illustrated in Figure 4.11.

**4.4.2 Persistent to Persistent - Related Patterns**

- **Proxy Data Mapper pattern**: must be applied before the application of this pattern.

![Diagram of Persistent-to-Persistent pattern](image)

Figure 4.11: Structure introduced by the *Persistent-to-Persistent* pattern
4.5 Persistent to Multi Persistent

Context  An instance of a persistent class is a client of multiple instances of another persistent class.

This pattern applies to cases where a single instance of a persistent class references multiple instances of another persistent class using some collection implementation. For example, in the e-mail client application example, each MailServer object may contain a collection of Message instances to be sent as illustrated in Figure 4.12. The context of the Persistent-to-Multi-Persistent is identified by the following combination:

<table>
<thead>
<tr>
<th>Navigation</th>
<th>Cardinality</th>
<th>Context class</th>
</tr>
</thead>
<tbody>
<tr>
<td>outgoing</td>
<td>*</td>
<td>persistent</td>
</tr>
</tbody>
</table>

Problems that arise in the context of the Persistent-to-Multi-Persistent pattern: The context of the Persistent-to-Multi-Persistent pattern leads to problems that are caused by the need to realize a duplicated collection that resides both in-memory and in-storage. The pre-condition of this pattern is that the 1 : many cardinality constraint is already implemented in the storage, probably using a standard database modeling tool, e.g., as a Primary-to-Foreign key relation in a Relational database implementation.

1. Persistent collection realization:

   The problem is to provide an in-memory realization for a 1 : many cardinality constraint,
where the referenced entities are persistently stored. The collection’s in-memory existence should be reduced to minimum, since its content is already stored. Therefore, only elements that are necessary for collection management should be realized, leaving the real content in storage.

2. **Collection creation problem:**
   Since both the persistent class and the collection are duplicated between memory and storage, whenever loading the fullweight representative of the persistent class, it should also be re-connected to its corresponding in-memory collection. The fullweight object should be oblivious to the fact that the collection introduced to it has been duplicated from storage.

3. **Collection load problem:**
   Whenever loading a collection object from storage to memory, it must be equivalently identified in-memory and in-storage. Since there are different means for identifying a collection in-memory and in-storage (due to different structures), the collection in-memory existence must be able to resolve its containment in storage. This containment identifier must be provided for each in-memory collection being loaded.

4. **The deletion problem:**
   The same problem as described in the *Persistent Collection Data Mapper* pattern arises here. Since the collection manages all of its containment objects in storage, their storage deletion procedure must be deprecated when they are still required by the collection.

**Problem solutions:**

1. **Solution to the persistent collection realization:** A specific collection, e.g., *MessageCollectionMapperImp*, for persistent elements of the context class is introduced. The elements of the new collection do not reside in-memory. The collection is responsible for supporting application level requests for in-memory copies of stored context class objects. It also provides an implementation for self deletion of all contained elements in storage when there are
no more in-memory references leading to it (e.g., `finalize` method implementation).

For example, when a full in-memory copy of a `MailServer` stored object requests a specific message, `MessagesCollectionMapperImp` loads an in-memory (proxy) copy of the requested `Message` instance. When the `MailServer` in-memory copy removes a message from its collection, `MessagesCollectionMapperImp` applies its deletion procedure.

2. **Solution to the collection creation problem:**

The responsibility to create the new specific collection and to attach it to the persistent class object is imposed on the `DataMapper` of the persistent class.

Similar to the `Persistent Collection Data Mapper` pattern, here as well the persistent class does not change, as the new collection is hidden being a subtype of a regular library collection class (e.g. `List`) and constructed using a self contained factory method.

This hiding is carried out by a factory method that replaces the construction of the in-memory collection with the construction of the specific collection. A pre-condition for this pattern is that factory methods exist not only for persistent classes but also for all collection implementations.

3. **Solution to the collection load problem:**

Since the `DataMapper` of the persistent class is responsible for the specific collection creation, it is also given the responsibility to provide it with the collection identifier in-storage.

For example, in a `Relational` database, the collection is identified using a `foreignKey` that has the same value of the corresponding `primaryKey`. Therefore, in a `Relational` storage, the id of the persistent object may be used to identify the collection of context objects in storage (illustrated in Figure 4.13). In any alternative data layer implementation, the datamapper is provided with a `resolving function` that identifies the collection id given the id of its owner.

A possible implementation for the load method of such datamapper is illustrated in Listing 4.2.
4.5. Persistent to Multi Persistent

**Figure 4.13: 1:M Implementation in a Relational database**

<table>
<thead>
<tr>
<th>MailServers</th>
<th>Messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>primaryKey</td>
<td>primaryKey</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>

**Listing 4.2: DataMapper load implementation that restores a collection reference**

```java
public class MailServerDataMapper {

    public MailServerImp load (int mailServerID) {
        MailServerImp result;
        String query = "SELECT * FROM MailServers WHERE serverID = " + mailServerID + " ;";
        Statement stmt;
        try {
            openConnection();
            stmt = con.createStatement();
            ResultSet res = stmt.executeQuery(query);
            res.next();
            String url = res.getString("url");
            result = new MailServerImp(url);
            //Attaches the collection with the given mailServerID
            result.setOutgoingFolder(MessagesCollectionMapper.create(mailServerID));
            res.close();
            closeConnection();
        } catch (SQLException ex) {
            System.out.println("SQLException: " + ex.getMessage());
            result = new MailServerImp("New Empty mail server created");
        }
        return result;
    }
}
```

4. **Solution to the deletion problem:** As in the Persistent Collection Data Mapper pattern, the new collection communicates with the table manager of the context class, in order to keep the reference count to stored context elements updated.
4.5. Persistent to Multi Persistent

4.5.1 Persistent to Multi Persistent - Detailed Description

4.5.1.1 Persistent to Multi Persistent - Input

Similar to the persistent-to-persistent pattern, both classes are persistent. The class that holds the reference (e.g., MailServer) is referred to as the persistent class and the referred class (e.g., Message) is referred to as the context class. Therefore, the application of this pattern requires the following inputs:

1. A persistent class referred to as the persistent class.
2. A persistent class referred to as the context class.

4.5.1.2 Persistent to Multi Persistent - Pre-condition

1. The post-condition of the Proxy-Data-Mapper pattern holds with respect to both classes the context and the persistent.
2. The persistent class is associated with the collection class.
3. The collection class implements a factory-method to support self instantiation.
4. The 1:many relationship between instances of the persistent and context classes is realized in the Data layer. Resolving function that accounts for the identification of the collection related to the persistent object is provided.

4.5.1.3 Persistent to Multi Persistent - Post-condition

1. A table-manager class exists to guard instance storage deletion of the context class.
2. A new persistent-collection class exists to realize the collection of persistent objects.
3. Both the persistent-collection and the context-proxy classes are associated with the table-manager class.
4.5.1.4 Persistent to Multi Persistent - Action

The action of the Persistent to Multi Persistent pattern is composed of sequential application of the following Transformation Building Blocks described in chapter 8:

1. **Introduce TableManager (problem 4):**
   Introduce a new tableManager to the contextClass. This transformation is the same as the first two transformations in the Persistent Collection Proxy Data Mapper pattern action detailed in section 4.1.1.4:
   
   ```java
   IntroduceTableManager(contextClass)
   ```

2. **Introduce persistent-collection (problem 1):**
   Introduce the new persistent collection to handle instances of the contextClass and to notify the tableManager about reference changes. The creation of the persistent collection also revises the factory method of the in-memory collection class:
   
   ```java
   collection = createP2PCollectionClass(collection, ctxtProxy, tableManager)
   ```

3. **Revise the dataMapper to load the collection(problems 2,3):**
   Assign the datamapper of the persistentClass to re-attach the fullweight representative of the persistent class with its corresponding persistent collection object:
   
   ```java
   restoreCollectionReferenceInLoadMethod(persistentClass.mapper, collection);
   ```

The Persistent to Multi Persistent pattern application result is illustrated in Figure 4.14.

4.5.2 Persistent to Multi Persistent pattern - Related Patterns

- **Persistent Collection Data Mapper pattern:** If the reference realization between the persistent class and the context class does not exist in the Data layer, the Persistent Collection Data Mapper pattern should be applied. In such case, the precondition of this pattern fails.

- **Proxy Data Mapper pattern:** must be applied before the application of this pattern.
Figure 4.14: Structure introduced by the Persistent-to-Multi-Persistent
Listing 4.1: Memoization Proxy Data Mapper - Revised MessageProxy class

```java
public class MessageProxy extends Message {
    private int ID;
    private MessageDataMapper dataMapper;
    private EncryptionKey key;

    public MessageProxy(String title) {
        dataMapper = MessageDataMapper.getUniqueInstance();
        MessageImp imp = new MessageImp(title);
        ID = dataMapper.store(imp);
        key = imp.key;
    }

    protected void finalize() throws Throwable {
        dataMapper.delete(ID);
    }

    private MessageImp load() {
        MessageImp imp = dataMapper.load(ID);
        imp.key = key;
        return imp;
    }

    private void update(MessageImp mes) {
        dataMapper.update(mes, ID);
        key = mes.key;
    }

    // Delegated methods:

    public void addAttachment() {
        MessageImp mes = load();
        mes.addAttachment();
        update(mes);
    }

    public String getBody() {
        Message mes = load();
        String result = mes.getBody();
        update(mes);
        return result;
    }

    ... more delegated methods
}
```
Chapter 5

Towards Automatic Insertion of the Data Access Layer

"Beware of bugs in the above code; I have only proved it correct, not tried it."

Donald Ervin Knuth

5.1 Data Access Layer insertion correctness

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 pattern require the existence of a proxy and a datamapper classes for the persistent class).

2. **Memoization excludes the PCDM application:**

   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 3.2, **PCDM** should not be applied to the **ScanSet** collection of **Message** objects because the memoization patterns apply to the contexts \( \langle Message, EncryptionKey \rangle \). The reason for this constraint is that the **PCDM** pattern replaces the lightweight proxy representatives of stored objects by their IDs. But, when the memoization patterns apply, the proxy objects include, in addition to the IDs, also the outgoing references to the memory objects. The application of the **PCDM** pattern loses these references.

3. **PCDM does not follow Persistent to Multi-Persistent:**

   In a context that satisfies the **Persistent to Multi-Persistent (P2MP)** pattern (and therefore includes a persistent collection), the **PCDM** pattern should not be the last to apply. Preferably, it should not apply. For example, in the email system in Figure 3.2, **PCDM** should not be applied to the **outgoingFolder** collection of **Message** objects because the **P2MP** pattern applies to the context \( \langle mailServer, Message \rangle \). 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 the **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.

In this chapter we introduce a non-deterministic algorithm for **DAL** insertion, that involves ap-
5.1. Data Access Layer insertion correctness

Application of all DAL patterns. The non-determinism reflects decisions that have no impact on the final result (although they might affect issues like performance, efficiency, and any-time properties). The deterministic decisions involve inter-pattern dependency constraints.

An algorithm requires a study of its correctness and properties. Correctness requires the specification of properties that must be fulfilled by the algorithm under consideration. Indeed, we would like to claim that the DAL insertion algorithm that we provide is correct, that is, fulfills the requirements of partial persistency insertion. But what exactly are these requirements? Overall, there are three major requirements: partial persistency, Domain-Data layers 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. Inter-layer integration means integration of existin layers, i.e., minimal changes to the existing layers. Certainly, an integration that replaces most of the domain layer classes with new ones is not acceptable. Our feeling is that an acceptable integration might involve, at most, changes to the integrated classes (the classes marked as persistent) and to their domain layer clients.

The DAL insertion algorithm is a code restructuring transformation, i.e., a refactoring. Therefore, it should satisfy the general requirement of refactorings, which is behavior preservation. This requirement is the hardest one to formalize. In simple formal computation models, it means equivalence of the tool, 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. This work suggests observables for the preservation of structure, inter-object reference, object availability and method computation. These observables serve as a correctness specification: A procedure that satisfies the observables is considered correct. The observables that this work adopt as criteria for Domain – Data Layers integration are as follows:

1. Partial persistency: All and only instances of classes marked as Persistent are stored in
durable storage.

2. **Transparent Domain-Data layers integration**: Minimize Domain layer changes.

3. **Object structure preservation**:
   
   (a) **Domain-Data Layers 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.

**Definition 1**

1. A **DAL insertion algorithm** is correct for Domain-Data integration if its application always satisfies the above observables.

2. A set of **DAL patterns** is complete if there exists a correct DAL insertion algorithm that is based on combination of patterns from the set.

3. A complete set of **DAL patterns** is minimal if the removal of a pattern from the set turns it incomplete.
5.2 Dal-Insertion Algorithm

Given any system with partial persistency requirements and its corresponding data-layer mapping specification, the Dal-Insertion algorithm for the combined procedure of patterns application is defined by Algorithm 1.

**Algorithm 1 Dal-Insertion algorithm**

```plaintext
procedure Dal-Insertion(PersistentClasses, PersistentCollections)

1. Apply Proxy-Data-Mapper for all PersistentClasses.

2. Non-deterministically apply the following:
   (a) Apply Memoization-Proxy-Data-Mapper to all appropriate context.
   (b) Apply Multi-Memoization-Proxy-Data-Mapper to all appropriate context.
   (c) Apply Persistent-to-Persistent to all appropriate context.
   (d) Apply Persistent-to-Multi-Persistent to all appropriate context.

3. Optional: Apply Persistent-Collection-Data-Mapper to collection of persistent classes \( P \) such that: (1) \( P \) does not participate in a context of a memoization pattern (i.e., excluding all collection 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
```

**Anytime variation on 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 that are already fully integrated with the data layer.

**Claim.** The introduced DAL-insertion algorithm is correct.

The claim holds since it can be shown that an application of the algorithm indeed satisfies all of the
above observables. In particular, the transparent integration requirement holds since the *Domain* layer stays intact. The duplication observables are satisfied by the individual patterns; reference preservation is handled by the memoization patterns, and object availability is guaranteed by the proxy objects 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 operation. Therefore, it is a correct algorithm for integration of the *Domain-Data Layers*.

**Claim.** 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.
5.3 Overall Application Example

This example demonstrates the application of the Data-Access patterns to the email system illustrated in Figure 3.2. The patterns require that collection classes are implemented using wrappers, and that all persistent classes and collections have factory methods.

The example applies the Data-Access patterns following the steps of Dal-Insertion algorithm.

**Application procedure begins here:** The input for the application with all in-memory collections is illustrated in Figure 5.1. All collections implement a List interface.

![Figure 5.1: Application Example - Input system](image)

1. **Application of the Proxy Data Mapper (PDM) pattern:**

   The PDM pattern is applied with respect to all classes that are marked persistent in the system. This entails the following applications:

   ProxyDataMapper(Message,"MessageTable")
   ProxyDataMapper(MailServer,"MailServerTable")
   ProxyDataMapper(Attachment,"AttachmentTable")

   The application result is illustrated in Figure 5.2. Each persistent class is added three new data-access constructs: a datamapper, a proxy and an implementation class. All factory methods are revised to return instances of the implementation class.
2. **Application of the Memoization Proxy Data Mapper (MPDM) pattern:**

The MPDM pattern is applied with respect to all persistent classes that have an outgoing reference (reference attribute) to an in-memory object. This entails the following application:

\[ \text{MemoizationProxyDataMapper}(\text{Message, EncryptionKey}) \]

The application result is illustrated in Figure 5.3. The \textit{MessageProxy} class was revised to handle the memoization of an \textit{EncryptionKey}.

3. **Application of the Multi Memoization Proxy Data Mapper (MMPDM) pattern:**

The MMPDM pattern is applied with respect to all persistent classes that have an outgoing reference to a collection of in-memory objects. This entails the following application:

\[ \text{MultiMemoizationProxyDataMapper}(\text{Message, History}) \]

The application result is illustrated in Figure 5.4. The \textit{MessageProxy} class was revised to handle the memoization of an \textit{History} collection.

4. **Application of the Persistent to Persistent (P2P) pattern:**

The P2P pattern is applied with respect to all persistent-to-persistent relations. This entails
Figure 5.3: Application Example - Memoization Proxy Data Mapper result

the following application:

**PersistentToPersistent**(*Message, Attachment*)

The application result is illustrated in Figure 5.5. A new class *AttachmentsTableManager* is introduced and the class *MessageDataMapper* is assigned to it.

5. **Application of the Persistent to Multi Persistent (P2MP) pattern:**

The P2MP pattern is applied with respect to all persistent-to-multi persistent relations. This entails the following application:

**PersistentToMultiPersistent**(*MailServer, Message*)

The application result is illustrated in Figure 5.6. A *MessageTableManager* class is introduced to handle destruction of message objects. A new *MessagesCollectionMapperImp* persistent collection class is introduced. The mapper of the *MailServer* is assigned to the new collection and the factory method of the *OutgoingFolder* collection is revised to produce objects of the new collection.
6. Application of the Persistent Collection Data Mapper (PCDM) pattern:

Optionally, The PCDM pattern should be applied with respect to all persistent collections that are not yet handled by the application procedure. Referring to the illustrated example, the only persistent collection that was not yet handled is the \textit{ScanSet} collection. This collection handles instances of the persistent class \textit{Message} which also participates within a Memoization-Proxy-Data-Mapper pattern context (i.e. \textit{Message}::\textit{EncryptionKey}). This entails that the \textit{ScanSet} persistent collection doesn’t satisfy the first DAL-algorithm application constraint (i.e. Algorithm 1 - 3.1) and therefore is not applied.

**Application procedure terminates here:** The final result of the application procedure to the email system is illustrated in Figure 5.7. All new Data-Access layer constructs are marked in gray color.
5.3. Overall Application Example

Figure 5.5: Application Example - Persistent to Persistent result

Figure 5.6: Application Example - Persistent to Multi Persistent result
5.3. Overall Application Example

Figure 5.7: Application Example - Final result
Chapter 6

Patterns Execution Tool (PET) implementation

PET [4] is an experimental tool that implements the static aspect of data layer construction. The tool aims at providing a flexible platform for patterns application. Being developed as a spike within the framework of this research, the PET tool development experiment has enlighten the applicability of such an implementation within today’s immature technologies and standards.

After inspecting various existing implementations (i.e. Rational Rose, Magic-Draw, Visual-Paradigm, Argo-UML) and current standards, related to the notion of schema interchange (i.e. XML, XMI, MOF), the decision was to develop a standalone tool that would be on one hand, platform independent, and on the other, compatible with most of current CASE tools offered by industry. Therefore, all inputs and outputs for representing class diagrams in PET are expressed using XMI file format. Likewise, the selected language for the implementation was Java.

As illustrated in Figure 6.1, PET deals with two types of inputs:

1. A static model of a system upon patterns are to be applied (XMI).

2. A set of patterns, expressed using an internal, self-defined, XML based language.

The output is the model of the system given by the input revised by a chosen sub-set of the input
patterns.

Figure 6.2 shows the input and the output generated by PET for the application of the Proxy-Data-Mapper pattern (pattern representation in XML is given within the appendix of this work in section ??). Generating the input XMI file and the presentation of the result was achieved by the usage of Argo-UML tool.

6.0.0.1 PET Architecture

PET structure (illustrated in Figure 6.3) consists of the following elements:
1. **Pattern Application Client** This module is responsible for pattern selection and application. Since application algorithms may vary between different clients, the concrete strategy of the application and its coherent representation of each pattern is separated from other modules. All related interfaces and a concrete implementation for patterns sequential application strategy is provided.

2. **Class-Diagram transformations provider** Supports all class-diagram editing services. The module is separated into input/output handling module (*Converters*) and transformation services provider module (*Class-Diagram transformer*).
   
   (a) **Converters** This module handles import and export of both class-diagram schema and patterns. The module includes all relevant interfaces and concrete implementations for XMI schema processing and for patterns annotated in XML.

   (b) **Class-Diagram transformer** This is the core module of PET which provides all basic services related to pattern application. The implementation distinguishes between four different basic services:

   i. **Matching** - handles *pattern-matching* [28] of a single pattern context within a given schema. A concrete implementation driven by navigational analysis of class diagram is provided.

   ii. **Operator Extraction** - given a single match and a single pattern, generates a transaction of all operators to be applied.

   iii. **Transaction application** - given a schema and a single transaction, revise the schema by executing all transformation operators within the transaction.

   iv. **Validation** - given a transaction of operators, checks whether it is generally valid or valid upon a given schema.

   PET is designed to have a flexible architecture in which all modules are independent of each other. This architecture can easily transform to fit different types of application strategies, patterns properties and descriptive standards for schema interchange.
PET development experience had shown that current technologies and standards can be used to achieve pattern automation. All technical complexities may be bridged and the integration of all data access pattern into a real-world systems is an applicable notion.
Figure 6.3: PET components
Chapter 7

Conclusions and Future work

This framework introduces a set of independent *Data Access Patterns* that provide the missing link towards full automation of the *Business-Data* layers interaction. The framework provides a complete set of patterns that cover all persistency – in-memory interaction schemes.

In addition, the framework sets up a theoretical basis for pattern driven software automation. The suggested framework is applicable at design level, can be fully automated, and does not imply any modification to domain layer classes.

7.1 Recovery process

Recovery means that in some case of software or hardware failure, the system should be brought back to some consistent state. Having one persistency mechanism or another, the recovery process is driven by the business activity of the domain where the system is being used. For example, in a case of power outage, an on-line store may decide to reset all uncommitted sales that were entered prior to system’s failure. However, the same system being used within a physical store should be able to recover all last sales data since customers are still waiting next to the cashier.

Although persistent data serves as the major resource for recovery, all concrete recovery procedures cannot be anticipated by any persistency handling system. It is the developer’s own re-
sponsibility to provide concrete recovery procedures for each memory restoring scenario. Thus, the recovery is non relevant to the scope of this work.

### 7.2 Transactions

Whenever dealing with a database system, the notion of a transaction arises. Most existing tools related to persistency, already provide transaction support (see an example for transaction support by Hibernate in Figure 7.1). For a system to support this notion, it must fulfill the following four properties of a transaction (ACID) [23]:

1. **Atomic operations**: The execution of a transaction should be considered as atomic. Either all operations within the transaction are carried out or none are.

2. **Consistency**: Consistency with the database must hold after transaction has been executed.

3. **Isolation**: Either concurrency is prevented or each transaction should be protected from the effects of other concurrently executed transformations.

4. **Durability**: The system must be committed for each transaction, that is, the results of each transaction should be persistent.

The framework introduced in this work supports all last three properties since consistency with the storage holds in all time, concurrency is avoided and the responsibility for durability is imposed on the storage management mechanism. However, referring to the structure of the core Proxy-Data-Mapper pattern, each action delegated by the proxy is immediately propagated into storage (Auto Commit).

In order to support wider extent of atomic operations, some transaction aggregation feature, similar to the one supported by Hibernate may be added to the solution framework. However, such feature has been out of scope of this work since the invocation of such mechanisms requires client interference which spoils the transparency of persistency handling insertion.
7.3 Performance

The common belief is that there is a trade-off between performance and persistency. Although storage management systems may have special algorithms and structures for handling the efficiency of large stored data, in-memory structures still achieve better performance. Thus, whenever exposing a system to additional persistency requirements, one should expect performance impacts.

Nevertheless, in enterprise systems, an optimization may be achieved not only by improving the performance of the persistent layer, but also by reducing the coupling between the domain and data layers as much as possible. Implementing various constructs (i.e. identity-map and lazy-load [9]) as part of the domain layer design may minimize reduction in performance.

In the same way, the architecture of the core pattern suggested in this framework, separates between performance (imposed in the datamapper) and persistency handling (imposed on the proxy). Therefore, performance tuning can be handled locally by various mapper implementations (i.e. a datamapper with “smart-loader” mechanism is shown in the appendix 9.1).

However, since the main motivation for this work was just filling the missing link between the domain and data layers, whenever thoughtful manual programming is required, performance analysis has not been within this scope. Nonetheless, in such critical systems, where performance is an essential matter, tuning up a concrete mapper and testing its performance is inevitable so to
minimize the effect of persistency handling.

7.4 Future Work

Current implementation of the framework in the experimental PET tool reflects only the static aspect of patterns application. Further development involves:

1. Integrating the proposed solution into a development environment (e.g. Eclipse) to reflect both static and dynamic aspects of the patterns, and their application within the context of a real-world Enterprise application and its corresponding storage system.

2. Investigating the possibility of using FLogic for internal symbolic representation of class diagrams, and for rule application. This would imply extension of the current PET implementation that uses a direct internal representation.

3. Providing an improved pattern selector module to in order to achieve complete equivalence with the combined procedure for patterns application as suggested by DAL-Algorithm.

4. Extending the implementation to support visual expression of pattern rules.

5. Embedding the resulting tool (as a stand alone component) in commercial CASE tools, so to enable full automation of persistency requirements.

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.
Bibliography


Chapter 8

Appendix A - Transformation Building Blocks

This section lists all *conditional transformations*, that serve as the building blocks for the refactorings that compose each of the Data-Access patterns. Each conditional transformation may be either *primitive* or *composite*.

A *primitive transformation* is a refactoring that is supported by the reflective mechanism of the environment (e.g. Eclipse, intelliJ etc.). All *primitive transformations* are listed in section 8.3 of this Chapter.

A *composite transformation* is like a transaction of conditional transformations either completed (commit) or rolled back. *Composite transformations* are split to two:

- **DAL specific transformations**, that describe transformations that refer to the global motivation of this work to introduce persistency requirements. These transformations are listed in section 8.1 of this Chapter.

- **General purpose transformations**, that describe general transformations that may be applied on any system’s structure. These transformations are listed in section 8.2 of this Chapter.
8.1 DAL specific transformations

The following tree structure illustrates the catalog of all DAL transformations. The transformations are categorized into different contexts of their application. Each of the DAL transformations is detailed below.

8.1.1 loadImp

Usage: ProxyDataMapper

Motivation: Generates a method that duplicates an identified persistent object from the Data-Layer into memory. The generated load method implementation constructs the fullweight object based on a corresponding Data-Layer element.

Arguments:

- className :: persistentClass [required]
- DBschemaElement :: schemaElement [required]
8.1. DAL specific transformations

Precondition:

- Class with name “persistentClass” exists.
- Element with name “schemaElement” exists in the Data-Layer.

Mechanics:

```java
result = initializeConnectionExpression(schemaElement);

for each property p in properties(persistentClass)
    result = concat(result, retrievePropertyExpression(p, schemaElement::arg::'arg ');

result = concat(result, terminateConnectionExpression(schemaElement));

return result;
```

Example: `loadImp("Message","[Table]:[Messages]")`

Result illustrated in Listing 8.1

Listing 8.1: Load method implementation

```java
public MessageImp load(int messageID) {
    MessageImp result;
    String query="SELECT * FROM Messages WHERE messageId =" + messageID + ";
    Statement stmt;
    try {
        openConnection();
        stmt = con.createStatement();
        ResultSet res = stmt.executeQuery(query);
        res.next();
        String subject = res.getString("subject");
        String body = res.getString("body");
        result = new MessageImp(subject);
        result.setBody(body);
        res.close();
    }
    finally {
        closeConnection();
    }
    return result;
}
```
8.1. DAL specific transformations

```java
    closeConnection();
}

    catch (SQLException ex) {
        System.out.println('SQLException: ' + ex.getMessage());
        result = new MessageImp('New Empty message created');
    }
    return result;
}
```

8.1.2 storeImp

**Usage:** ProxyDataMapper

**Motivation:** Generates a method that encapsulates persistency services for duplicating a persistent object into the Data-Layer from memory. The method is executed through the mapping between the persistent class and its corresponding representation within the persistent layer.

The duplication into the Data-Layer also generates an identifier for the persistent object representation in storage.

**Arguments:**

- className :: persistentClass [required]
- DBschemaElement :: schemaElement [required]

**Precondition:**

- Class with name “persistentClass” exists.
- Element with name “schemaElement” exists in the Data-Layer.

**Mechanics:**

```java
    result = generateIDexpression(persistentClass); // ID = ...

    result = concat(result,
                    initializeConnectionExpression(schemaElement));
```
8.1. DAL specific transformations

for each property p in properties(persistentClass)
    result = concat(result,
        setPropertyExpression(p, value(p, arg), schemaElement::ID));

result = concat(result,
    terminateConnectionExpression(schemaElement));

return result;

Example: storeImp("Message","Table.[Messages]");

Result illustrated in Listing 8.2

Listing 8.2: Storing method implementation

```java
public int store(MessageImp mes) {
    int messageID = generateGUID();

    String subject = mes.getSubject();
    String body = mes.getBody();
    String query = "INSERT INTO Messages(subject, body, messageID) VALUES(" + subject + ", " + body + ", " + messageID + "); ";

    try {
        openConnection();
        stmt = con.createStatement();
        int res = stmt.executeUpdate(query);
        System.out.println("Number of inserted rows: " + res + " with ID: " + messageID);
        stmt.close();
        closeConnection();
    }
    catch (SQLException ex) {
        System.out.println("SQLException: "+ex.getMessage());
    }
    return messageID;
}
```

8.1.3 updateImp

Usage: ProxyDataMapper
Motivation: Generates a method that encapsulates persistency services for updating the state of a persistent object from memory to its duplication within the persistent layer.

Arguments:

- className :: persistentClass [required]
- DBschemaElement :: schemaElement [required]

Precondition:

- Class with name “persistentClass” exists.
- Element with name “schemaElement” exists in the Data-Layer.

Mechanics:

```java
result = initializeConnectionExpression(schemaElement);
for each property p in properties(persistentClass)
  result = concat(result, updatePropertyExpression(p, value(p, arg[0]),schemaElement::arg[1]));
result = concat(result, terminateConnectionExpression(schemaElement));
return result;
```

Example: updateImp(“Message”,“Table.[Messages]”);

Result illustrated in Listing 8.3

Listing 8.3: Update method implementation

```java
public void update(MessageImp mes, int messageID) {
  String query="UPDATE Messages SET ";
  query+="subject = "+mes.getSubject()+", ";
  query+="body = "+mes.getBody()+""
```

query=$\text{WHERE messageID}$=$\text{messageID}$$\text{; ;}$

Statement stmt;

try {
  openConnection ( ) ;
  stmt = con . createStatement ( ) ;
  int res = stmt.executeUpdate(query);
  System . out . println ( res +"Message with ID: $\text{messageID}$ updated. ");
  closeConnection ( ) ;
}

catch (SQLException ex) {
  System . out . println ( "SQLException: $\text{ex}.$getMessage ()" );
}

8.1.4 deleteImp

Usage: ProxyDataMapper

Motivation: Generates a method that encapsulates persistency services for the deletion of a persistent object from the persistent layer.

Arguments:

- className :: persistentClass [required]
- DBschemaElement :: schemaElement [required]

Precondition:

- Class with name “persistentClass” exists.
- Element with name “schemaElement” exists in the Data-Layer.

Mechanics:

\[
\text{result} = \text{initializeConnectionExpression} (\text{schemaElement}) ;
\]

\[
\text{result} = \text{concat} (\text{result} ,
\text{deleteFromStorageExpression} (\text{schemaElement}) ) ;
\]
8.1. DAL specific transformations

```java
result = concat(result,
    terminateConnectionExpression(schemaElement));
return result;
```

**Example:** deleteImp(“Message”,“Table.[Messages]”);

Result illustrated in Listing 8.4

```java
public void delete(int messageID) {
    String query="DELETE FROM Messages WHERE " + messageID + " ;"
    Statement stmt;
    try {
        openConnection();
        stmt = con.createStatement();
        int res = stmt.executeUpdate(query);
        System.out.println(res+"Message with ID:"+messageID+" deleted.");
        closeConnection();
    }
    catch (SQLException ex) {
        System.out.println("SQLException:",ex.getMessage());
    }
}
```

8.1.5 **notifyToTableManagerOnDelete**

**Usage:** PersistentToPersistent

**Motivation:** When the data-mapper is requested to delete a persistent object from the persistent layer, the mapper should notify all table-managers of all related persistent objects. This transformation enriches the mapper class with corresponding call to a specific table-manager to decrease its reference count by 1.

**Arguments:**
8.1. DAL specific transformations

- className persClsMpr [required]
- className tblMgr [required]

Precondition:

- Class with name “persClsMpr” exists.
- Class with name “tblMgr” exists.

Mechanics:

```java
let: persistentClass = persClsMpr.getSuperName();
    contextClass = tblMgr.getCtxtName();

load = concat(persistentClass+"Imp me = load();");

getRef = concat(contextClass + "Proxy ref = (" + contextClass + "Proxy) me.get" + contextClass + "();");
cond = "if ( ref!=null )";
notify = concat(tblMgr + ".getUniqueInstance().decrease(ref.getId());");

oldDelete = getMethodImplementation(persClsMpr,"delete (...)" );
newDelete = concat(oldDelete + load + getRef + cond + notify);
addMethodImplementation(persClsMpr,"delete (...)",newDelete);
```

Example: The execution result of:

`notifyTableManagerOnDelete(MessageDataMapper,attachmentsTableManager)`

is detailed in Listing 8.5.

Listing 8.5: MessageDataMapper notifies to attachmentsTableManager on deletion

```java
public void delete(int messageId) {
    //Propagate deletion to table managers
    MessageImp msg = load(messageId);
    AttachmentProxy att = (AttachmentProxy) msg.getAttachment();
    if (att!=null)
```
8.1.6 notifyToTableManagerOnUpdate

**Usage:** PersistentToPersistent

**Motivation:** When the data-mapper is requested to update the state of an in-memory persistent object into the persistent layer, any changes in its outgoing references should be notified to all corresponding table-managers. This transformation enriches the mapper class with a call to a specific table-manager to update its reference count accordingly.

**Arguments:**

- `className persClsMpr [required]`
- `className tblMgr [required]`

**Precondition:**

- Class with name “persClsMpr” exists.
- Class with name “tblMgr” exists.

**Mechanics:**

```java
let persistentClass = persClsMpr.getSuperName();
contextClass = tblMgr.getCtxtName();

notify_decrease = concat(tblMgr + ".getUniqueInstance().decrease(
    id of instance in storage);"));
notify_increase = concat(tblMgr + ".getUniqueInstance().increase(
    id of instance in-memory);"));
```
8.1. DAL specific transformations

cond = "if (ref!=null) then";
option_true = "if hasRefferenceInDb()==TRUE then " + notify_increase;
option_false = "if hasRefferenceInDb()==FALSE then " + notify_decrease;

addHasReferenceInDb(persClsMpr);
oldUpdate = getMethodImplementation(persClsMpr,"update (...) ");
getRef = concat(contextClass + "Proxy ref = (" + contextClass + "Proxy
" + arg + ").get" + contextClass + ");

newUpdate = concat(oldUpdate + getRef + cond + option_true
+ option_false);
addMethodImplementation(persClsMpr,"update (...) ",newUpdate);

Example: The execution result of:

notifyTableManagerOnUpdate(MessageDataMapper,attachmentsTableManager)
is detailed in Listing 8.6.

Listing 8.6: MessageDataMapper notifies to attachmentsTableManager on update

```java
public class MessageDataMapper extends Message{

private void update(Message mes) {

  AttachmentProxy att=(AttachmentProxy) mes.getAttachment();
  if (att!=null) {
    int attid=att.getID();
    if (hasAttachmentInDB()==0)
      AttachmentsTableManager.getUniqueInstance().increase(attid);
  } else {
    int attid=hasAttachmentInDB();
    if (attid!=0)
      AttachmentsTableManager.getUniqueInstance().decrease(attid);
  }
}
```
8.1.7 notifyToTableManagerOnStore

Usage: PersistentToPersistent

Motivation: When the data-mapper is requested to store a new persistent object into the persistent layer, all of its corresponding table-managers should be notified about the creation. This transformation enriches the mapper class with a call to a specific table-manager to increase its reference count by 1 when the store method is invoked.

Arguments:

- className persClsMpr [required]
- className tblMgr [required]

Precondition:

- Class with name “persClsMpr” exists.
- Class with name “tblMgr” exists.

Mechanics:

```java
let: persClass = persClsMpr.getSuperName();
    contextClass = tblMgr.getCtxtName();

    notify_increase = concat(tblMgrName + 
                            ".getUniqueInstance().increase(ref.getId());");

    cond = "if (ref!=null) then";

    oldStore = getMethodImplementation(persClsMpr,"store (...)");
    getRef = concat(contextClass + "Proxy ref = (" + contextClass + "Proxy"
                    "+ arg +".get" + contextClass + ");");
    newStore = concat(oldStore + getRef + cond + notify_increase);
    addMethodImplementation(persClsMpr,"store (...)",newStore);
```
8.1. DAL specific transformations

**Example:** The execution result of:

```
notifyTableManagerOnStore(MessageDataMapper, attachmentsTableManager)
```

is detailed in Listing 8.7.

Listing 8.7: MessageDataMapper notifies to attachmentsTableManager on store

```java
public class MessageDataMapper extends Message{
    ...
    private void store(MessageImp mes){
        ...
        AttachmentProxy att=(AttachmentProxy) mes.getAttachment();
        if (att!=null)
            AttachmentsTableManager.GetInstance().increase(att.getID());
    }
}
```

8.1.8 updateStorageRefInUpdateMethod

**Usage:** PersistentToPersistent

**Motivation:** When the data-mapper is requested to update the state of an in-memory persistent object into the persistent layer, any changes to its outgoing references to other persistent objects should be updated in storage as well. This transformation enriches the data-mapper with such functionality corresponding to a single outgoing reference to another persistent class.

**Arguments:**

- className persMpr [required]
- className ctxtCls [required]

**Precondition:**

- Class with name “persMpr” exists with update() method.
- Class with name “ctxtCls” exists.
8.1. DAL specific transformations

Mechanics:

```java
let: persClass = persMpr.superClassName();

getRef = concat(ctxtCls + "Proxy ref = (" + ctxtCls + "Proxy)" + arg + ").get" + ctxtCls + ")(;");
hasReferenceCondition = "if \( \text{ref!} = \text{NULL} \)";
otherwise = "else";
executeQuery = "//Execute the query...";
queryTrue = concat("String query = "UPDATE " + persClass + "}s SET " + ctxtCls + "}" ref \. getID () \} WHERE \text{ID} = \text{ID}';");
queryFalse = concat("String query = "UPDATE " + persClass + "}s SET " + ctxtCls + "} = \text{NULL} WHERE \text{ID} = \text{ID}';");

oldUpdate = getMethodImplementation(persMpr,"update (...)");
newUpdate = concat(getRef + hasReferenceCondition + queryTrue + otherwise + queryFalse + executeQuery + oldUpdate);
addMethodImplementation(persMpr,"update (...)",newUpdate);
```

Example: The execution result of:

```
updateStorageRefInUpdateMethod(MessageDataMapper,Attachment)
```

is detailed in Listing 8.8.

Listing 8.8: MessageDataMapper updates storage according to linked instance

```java
private void update(Message mes) {
    String query;
    AttachmentProxy att=(AttachmentProxy) mes.getAttachment();
    query = "UPDATE,Messages,SET";
    ...
    if (att != null)
    {
        query+= "attachment=",+att.getID()+"";
    }
    else
    {
        query+= "attachment=NULL";
    }
    // Executes the query
}
```
8.1.9 updateStorageRefInStoreMethod

**Usage:** PersistentToPersistent

**Motivation:** Similar to `updateStorageRefInUpdateMethod` transformation, when the data-mapper is requested to store a new in-memory persistent object into the persistent layer, any existing outgoing references to other persistent objects should be stored into storage as well. This transformation enriches the data-mapper with such functionality corresponding to a single outgoing reference to another persistent class.

**Arguments:**

- `className persMpr` [required]
- `className ctxtCls` [required]

**Precondition:**

- Class with name “`persMpr`” exists with store() method.
- Class with name “`ctxtCls`” exists.

**Mechanics:**

```java
let: persClass = persMpr.superClassName();

getRef = concat(ctxtCls + "Proxy ref = (" + ctxtCls + "Proxy"
" + arg + ".get" + ctxtCls + ");");
hasReferenceCondition = "if (ref!=NULL)"
otherwise = "else";
executeQuery = "//Execute the query...";
queryTrue = concat("String query = 'INSERT INTO " + persClass
"s (... all columns...) VALUES ..." + ctxtCls.getName()
" = ref.getId ()' ;");
queryFalse = concat("String query = 'INSERT INTO " + persClass
"s (... all columns...) VALUES ..." + ctxtCls.getName()
```
8.1. DAL specific transformations

```java
oldUpdate = getMethodImplementation(persMpr,"store ( . . . ) ");
newUpdate = concat(getRef + hasReferenceCondition +
queryTrue + otherwise + queryFalse +
executeQuery + oldUpdate);
addMethodImplementation(persMpr,"store ( . . . ) ", newUpdate);
```

Example: The execution result of:

`updateStorageRefInStoreMethod(MessageDataMapper,Attachment)`

is detailed in Listing 8.9.

```
Listing 8.9: MessageDataMapper updates storage according to linked instance
private void store(MessageImp mes) {
    AttachmentProxy att=(AttachmentProxy) mes.getAttachment();
    String query;
    if (att!=null) {
        query = "INSERT INTO Messages (subject, body, messageID, attachment) VALUES("+subject+"," +body+"," +messageID+"," +att.getID();
    } else {
        query = "INSERT INTO Messages (subject, body, messageID) VALUES("+subject+"," +body+"," +messageID+");"
    }
    // Executes the query
}
```

8.1.10 restoreRefInLoadMethod

Usage: PersistentToPersistent

Motivation: When the data-mapper is requested to load a persistent object from the data-layer, it
should also reconnect it to all related persistent objects as appears within the data-layer. This trans-
f ormation enriches the data-mapper with such functionality to restore a single outgoing reference of
a loaded persistent object.
8.1. DAL specific transformations

Arguments:

- className persMpr [required]
- className ctxtCls [required]

Precondition:

- Class with name “persMpr” exists with load() method.
- Class with name “ctxtCls” exists.

Mechanics:

```plaintext
let getRefInStorage = concat("ref = has"+ctxtCls+"InDB();");
hasReferencerCondition = "if (ref!=0);"
reAttach = concat("result.set"+ctxtCls
 +"("+ctxtCls+"."create(ref));");

oldLoad = getMethodImplementation(persMpr,"load()");
newLoad = concat(oldLoad + getRefInStorage +
 hasReferencerCondition + reAttach);
addMethodImplementation(persMpr,"load()",newLoad);
```

Example: The execution result of:

```
restoreRefInLoadMethod(MessageDataMapper,Attachment)
```

is detailed in Listing 8.10.

```
Listing 8.10: MessageDataMapper re-attaches a linked instance

private MessageImp load(int messageID) {
  MessageImp result;
  ...
  // result instance has been loaded from storage
  int refId = hasAttachmentInDB(messageID);
  if (refId!=0)
    result.setAttachment(Attachment.create(refId));
  ...
  return result;
}
```
8.1.11 restoreColRefInLoadMethod

**Usage:** PersistentToMultiPersistent

**Motivation:** Similar to the `restoreRefInLoadMethod` transformation, when the data-mapper is requested to load a persistent object from the data-layer, it should also reconnect it to all related persistent collection objects as appears within the data-layer. This transformation enriches the data-mapper with such functionality to restore a single outgoing reference of a loaded persistent object. The identifier of the loaded persistent object identifies the collection as well.

**Arguments:**

- className `persMpr` [required]
- className `collCls` [required]

**Precondition:**

- Class with name “persMpr” exists with load() method.
- Collection class with name “collCls” exists.

**Mechanics:**

```javascript
let: persClass = persMpr.getSuperName();
ctxtClass = collCls.getItemsName();
collectionRoleName = associationName(persClass, ctxtClass);

reattach = concat("result.set"+collectionRoleName
    +"("+ctxtClass+"sCollectionMapperImp.create(ID));");

oldLoad = getMethodImplementation(persMpr,"load (...) ");
newLoad = concat(oldLoad + reAttach);
addMethodImplementation(persMpr,"load (...) ",newLoad);
```
Example: The execution result of:

```java
class MailServerDataMapper {
    private MailServerImp load(int mailServerID) {
        MailServerImp result;
        ...  // Load
        result.setOutgoingFolder(MessagesCollectionMapper.create(mailServerID));
        return result;
    }
}
```

is detailed in Listing 8.11.

Listing 8.11: MailServerDataMapper re-attaches a collection instance

8.1.12 addDelegatedMethod

Usage: ProxyDataMapper

Motivation: Whenever a public method of a proxy class is invoked, the proxy is responsible to construct a new fullweight representative and delegate to its corresponding method implementation. This transformation adds a new delegated method to a 'proxy' class. The signature of the new method is identical to the given 'mSig' signature. The implementation of the the delegated method is a sequence of the following three instructions:

1. **Load** - calls the corresponding data-mapper to reconstruct the fullweight representative.

2. **Forward** - delegates the call to a method with the same signature in the fullweight representative. If the method has a returned value, it is kept within a temporary variable `result`.

3. **Store** - calls the corresponding data-mapper to save the state of the fullweight instance after the execution of the delegate method has terminated and returns `result` if exists.

Arguments:

- `className proxy` [required]
- `methodSignature mSig` [required]
8.1. DAL specific transformations

- className concrete [required]

Precondition:

- Class with name ’proxy’ exists.
- Class with name ’concrete’ exists.
- Class ’concrete’ implements a method with signature ’mSig’.

Mechanics:

```java
let: delegatedSignature = mSig;

addMethodSignature(proxy, delegatedSignature);
load = concat(concrete + " concrete = dataMapper.load(ID);");
if returnTypeOf(mSig) is not 'void'
    delegate = concat(ReturnType(mSig) + " result = concrete." + mSig + "(" + getParameters(mSig) + ");");
    store = concat("dataMapper.update(concrete.ID); + "return result;";");
else
    delegate = concat("concrete." + mSig +"(" + getParameters(mSig)+");");
    store = "dataMapper.update(concrete.ID);";

delegatedBody = concat(load + delegate + store);
addMethodImplementation(proxy, delegatedSignature, delegatedBody);
```

Example: Given the code in Listing 8.12, the execution result of:

```java
addDelegatedMethod(MessageProxy,MessageImp,getSubject())
```

is detailed in Listing 8.13.

**Listing 8.12: addDelegatedMethod - Before transformation**

```java
public class MessageProxy extends Message{
    ...
}
```
This implementation may be further optimized using a *smart-unloader* mechanism. This mechanism may determine whether to load or unload objects of the persistent class by monitoring their usage frequency. An example for such mechanism is detailed in Appendix-B of this work.

8.1.13  **reAttachReferenceInLoadMethod**

**Usage:** Memoization Proxy Data Mapper, Multi Memoization Proxy Data Mapper

**Motivation:** When the `load()` method of a proxy instance is invoked, if the loaded fullweight representative has an outgoing reference to an in-memory class, the proxy re-attaches the linked in-memory instance to the fullweight persistent object that is being constructed from storage.

**Arguments:**

- `method load [required]`

- `className ctxtCls [required]`

**Precondition:**

- Class with name ’ctxtCls’ exists.

- Method ’load’ exists in a proxy-class.
8.1. DAL specific transformations

Mechanics:

```java
let: persClass = method.getClass().getSuperClass();
    refName = associationName(persCls, ctxtCls);
    reAttach = concat("result.set"+ctxtCls+"("+refName+",");

    oldLoadImp = getMethodImplementation(load);
    newLoadImp = concat(oldLoadImp+reAttach);
    setMethodImplementation(load, newLoadImp);
```

Example: The execution result of:

```
reAttachReferenceInLoadMethod(MessageProxy.getLoadMethod(), EncryptionKey)
```

is detailed in Listing 8.14.

**Listing 8.14: MessageProxy re-attaches a linked in-memory instance in load() implementation**

```java
public class MessageProxy extends Message{
private EncryptionKey encrypt;

private MessageImp load(){
    MessageImp result = dataMapper.load(ID);
    result.setEncryptionKey(encrypt);
    return result;
}
```

8.1.14 updateReferenceValueInConstructor

Usage: Memoization Proxy Data Mapper, Multi Memoization Proxy Data Mapper

Motivation: Whenever new persistent object is constructed, if its fullweight representative after the construction process has an outgoing references to an in-memory class, the proxy also updates its corresponding reference value.

Arguments:
8.1. DAL specific transformations

- method constructor [required]

- className ctxCls [required]

Precondition:

- Class with name 'ctxCls' exists.

- Method 'constructor' exists in a proxy-class.

Mechanics:

```java
let: persClass = constructor.getClass().getSuperClass();
refName = associationName(persClass, ctxCls);
getRefMethod = concat("get"+ctxCls+"();");
updateRef = concat(refName+"=arg."+getRefMethod);

oldConstructorImp = getMethodImplementation(constructor);
newConstructorImp = concat(oldConstructorImp+updateRef);
setMethodImplementation(constructor, newConstructorImp);
```

Example: The execution result of:

`updateReferenceValueInConstructor(MessageProxy(Constructor1, EncryptionKey)` is detailed in Listing 8.15.

Listing 8.15: MessageProxy updates a reference attribute in Constructor() implementation

```java
public class MessageProxy extends Message{

private EncryptionKey encrypt;

protected MessageProxy(String title) {
    ...
    encrypt = mes.getEncryptionKey();
}

...}
```
8.1. DAL specific transformations

8.1.15 updateReferenceValueInUpdateMethod

Usage: Memoization Proxy Data Mapper, Multi Memoization Proxy Data Mapper

Motivation: Similar to the updateReferenceValueInConstructor transformation, whenever a persistent object is saved into the data-layer, if its fullweight representative has an outgoing references to an in-memory class, the proxy also updates its corresponding reference value.

Arguments:

- method update [required]
- className ctxtCls [required]

Precondition:

- Class with name 'ctxtCls' exists.
- Method 'update' exists in a proxy-class.

Mechanics:

```javascript
let persClass = constructor.getClass().getSuperClass();
refName = associationName(persClass, ctxtCls);
getRefMethod = concat("get"+ctxtCls+"();");
updateRef = concat(refName+"=arg."+getRefMethod);
oldUpdateImp = getMethodImplementation(update);
newUpdateImp = concat(oldUpdateImp+updateRef);
setMethodImplementation(update, newUpdateImp);
```

Example: The execution result of:

updateRefInUpdateMethod(MessageDataMapper, EncryptionKey)

is detailed in Listing 8.16.
Listing 8.16: MessageProxy updates a reference attribute in Update() implementation

```java
public class MessageProxy extends Message {

    private EncryptionKey encrypt;

    private void update(MessageImp arg) {
        encrypt = arg.getEncryptionKey();
    }
}
```

### 8.1.16 `createTableManager`

**Usage:** Persistent Collection Data Mapper

**Motivation:** This transformation creates a corresponding `TableManager` class for a given persistent class. The created class is responsible for handling reference count from various clients to each instance of the persistent class. When reference count to an instance of the persistent class reaches zero, the `TableManager` is responsible for deleting it (through a corresponding data-mapper) from the data-layer.

**Arguments:**

- `className persCls [required]`

**Precondition:**

- Class with name `persCls` exists.

**Mechanics:**

```java
let: name = concat(persCls + 'sTableManager');
    countedIdsField = "private CountedIds mappersCountedList";
    dataMapperField = "private " + persCls + "DataMapper dataMapper;";

    consSig = concat("private void " + persCls + "sTableManager()");
    consBody = "mappersCountedList = new CountedIds();\n```
8.1. DAL specific transformations 125

```
dataMapper = '" + persClass + '" + DataMapper.getUniqueInstance () ;" ;

increaseSig = "public void increase ( int id )" ;
decreaseSig = "public void decrease ( int id )" ;
increaseBody = " mappersCountedList . increaseCount ( id ) ;" ;
decreaseBody = " mappersCountedList . decreaseCount ( id ) ;
" + 
" if ( mappersCountedList . getCount ( id ) == 0 ) {
" + 
" dataMapper . delete ( id ) ;
" + 
" mappersCountedList . removeCount ( id ) ;} ;"

addClass ( name ) ;
addField ( name , countedIds ) ;
addField ( name , dataMapperField ) ;
addConstructor ( name , consSig ) ;
addMethodImplementation ( name , consSig , consBody ) ;
addMethodSignature ( name , increaseSig ) ;
addMethodImplementation ( name , increaseSig , increaseBody ) ;
addMethodSignature ( name , decreaseSig ) ;
addMethodImplementation ( name , decreaseSig , decreaseBody ) ;
```

**Example:** The execution result of:

```
createTableManager ( Message )
```

is detailed in Listing 8.17.

### Listing 8.17: Structure of MessagesTableManager Class

```
public class MessagesTableManager {
    // Implemented as Singleton
    private static MessagesTableManager sInstance = null;
    public static synchronized MessagesTableManager getUniqueInstance () {
        if ( sInstance == null ) {
            sInstance = new MessagesTableManager () ;
        }
        return sInstance ;
    }
    // Dual column table of id and its occurrences counting
    private CountedIds mappersCountedList ;
```
8.1. DAL specific transformations

private MessageDataMapper dataMapper;

// Constructor
private MessagesTableManager () {
    dataMapper = MessageDataMapper.getUniqueInstance();
    mappersCountedList = new CountedIds();
}

// increases the count for a given id
public void increase(int id) {
    mappersCountedList.increaseCount(id);
}

// decreases the count for a given id and deletes from
// storage if counter equals zero
public void decrease(int id) {
    mappersCountedList.decreaseCount(id);
    if (mappersCountedList.getCount(id) == 0) {
        dataMapper.delete(id);
        mappersCountedList.removeCount(id);
    }
}

8.1.17 assignTableManager

Usage: Persistent Collection Data Mapper

Motivation: This transformation assigns each constructor and each destructor/finalize method of a proxy class to notify the tableManager about creation and deletion of self instances.

Arguments:
- className proxy [required]
- className tableManager [required]

Precondition:
- Class with name 'proxy' exists.
- Class with name 'tableManager' exists.
Mechanics:

```java
let: increaseCall = concat(tableManagerName + 
    ".getUniqueInstance().increase(ID)");

decreaseCall = concat(tableManagerName + 
    ".getUniqueInstance().decrease(ID)");

for each element in constructors(proxy)
    oldBody = methodImplementation(element);
    newBody = concat(oldBody + increaseCall);
    addMethodImplementation(proxy, element, newBody);

oldFinalize = getMethodImplementation(proxy," finalize ()");
newFinalize = replace(oldFinalize,"dataMapper.deleteFromDB();",""); 
newFinalize = concat(newFinalize + decreaseCall);
addMethodImplementation(proxy," finalize ()",newFinalize);
```

Example: The execution result of:

```java
assignTableManager(AttachmentProxy,attachmentsTableManager)
```

is detailed in Listing 8.18. Each constructor contains a call to the table-manager’s increase method, and the finalize method contains a call to the table-manager’s decrease method. The previous invocation of `dataMapper.deleteFromDB()` is omitted.

### Listing 8.18: AttachmentDataMapper assigned to interact with a TableManager

```java
public class AttachmentProxy extends Attachment {

    private int ID;
    private AttachmentDataMapper dataMapper;

    protected AttachmentProxy(String filename, String path) {
        dataMapper = AttachmentDataMapper.getUniqueInstance();
        ID = dataMapper.store(new AttachmentImp(filename, path));
        AttachmentsTableManager.getUniqueInstance().increase(ID);
    }

    // Constructs a Proxy to an existing attachment in the DB
    public AttachmentProxy(int attid) {
        ID = attid;
        dataMapper = AttachmentDataMapper.getUniqueInstance();
        AttachmentsTableManager.getUniqueInstance().increase(ID);
    }
```
8.1. DAL specific transformations

private void update(AttachmentImp attachment) {
    dataMapper.update(attachment, ID);
}

protected void finalize() throws Throwable {
    AttachmentsTableManager.getUniqueInstance().decrease(ID);
}

private AttachmentImp load() {
    return dataMapper.load(ID);
}

// Delegated methods
public String toString() {...}
public int getID() {...}

8.1.18 createPersistentCollectionClass

Usage: Persistent Collection Data Mapper

Motivation: This transformation creates a new class that manages a collection of elements stored within the data-layer. The collection class implements a List interface. This transformation generates an in-memory collection that keeps an ID map of its contained elements in-memory. Whenever a new element is added, the collection keeps only the element’s ID. Whenever an element is requested, that collection constructs a corresponding lightweight representative using the ID of the retrieved element in the map. The collection notifies a tableManager object correlated with its managed items whenever items are added or removed. This transformation also redirects the factory method of the original collection given as an argument.

Arguments:

- className collection [required]
- className proxy [required]
8.1. DAL specific transformations

- className tableManager [required]

**Precondition:**

- A class with name 'collection' exists.
- A class with name 'proxy' exists.
- A class with name 'tableManager' exists.

**Mechanics:**

```java
let: persCls = proxy.getSuper().getName();
collName = concat(persCls+"sCollectionMapperImp");
mapField = "private indexedIds ids'";
consSig = concat("public "+collName+"()");
consBody = "ids = new indexedIds();"

addSig = "public void add(int index, Object element)"
addBody = concat("int elementID = ((
   +persCls+"Proxy\)element).getID();")
addBody+= "ids.add(index, elementID);"
addBody+= concat(tableManager+"...increase(elementID);")
removeSig = "public Object remove(int index)"
removeBody = "int elementId = ids.remove(index);"
removeBody+= concat(tableManager+"...decrease(elementId);")
removeBody+= concat("return (new "+proxy+"(elementId));")

getSig = "public Object get(int index)"
getBody = "int elementId = ids.resolveId(index);"
getBody+= concat("return (new "+proxy+"(elementId));")

sizeSig = "public int size()"
sizeBody = "return ids.size();"

addClass(collName);
addField(collName, mapField);
addConstructor(collName, consSig);
addMethodImplementation(collName, consSig, consBody);
```
addMethodSignature(collName, addSig);
addMethodImplementation(collName, addSig, addBody);

addConstructor(collName, removeSig);
addMethodImplementation(collName, removeSig, removeBody);

addMethodSignature(collName, getSig);
addMethodImplementation(collName, getSig, getBody);

addMethodSignature(collName, sizeSig);
addMethodImplementation(collName, sizeSig, sizeBody);

for each factoryMethod f in factories(collection)
  redirectFactoryMethod(collection, collName, f);

Example: The execution result of:

cREATEPersistentCollectionClass(ScanSet, MessageProxy, MessagesTableManager)

is detailed in Listing 8.19.

Listing 8.19: MessagesCollectionMapperImp persistent collection class implementation

```java
public class MessagesCollectionMapperImp extends MessagesCollectionMapper {

  // Map of IDs
  private IndexedIds ids;

  public MessagesCollectionMapperImp () {
    ids = new IndexedIds();
  }

  public void add(int index, Object element) {
    // gets element’s ID and drops it
    int elementId = ((MessageProxy)element).getID();
    ids.add(index, elementId);

    // notifies tableManager
    MessagesTableManager.getUniqueInstance().increase(elementId);
  }

  public Object remove(int index) {
    // removes ID from MAP
    int elementId;
    elementId = ids.remove(index);
  }
```
8.1.19 createP2PCollectionClass

**Usage:** Persistent To Multi Persistent

**Motivation:** This transformation creates a new class that manages a collection of elements that are stored within the data-layer. The collection class implements a `List` interface. This transformation generates a *persistent collection* that manages the collection of its contained elements and IDs within the data-layer. The ID of the collection serves as the identifier of the collection within storage (e.g., as the FK value to its owning instance in storage). Whenever an element is requested, that collection first retrieves its ID from the data-layer and then constructs a lightweight representative accordingly. The collection notifies a `tableManager` object correlated with its managed items whenever items are added or removed. This transformation also redirects the factory method of the original collection given as an argument.

**Arguments:**

- `className collection` [required]
- `className proxy` [required]
8.1. DAL specific transformations

- className tableManager [required]

Precondition:

- A class with name 'collection' exists.
- A class with name 'proxy' exists.
- A class with name 'tableManager' exists.

Mechanics:

```java
let: collName = concat(persCls+"sCollectionMapperImp");
consSig = concat("public List "+collName+"(int colId)");

addSig = "public void add(int index, Object element)";
addBody = concat("int elementID = (" +persCls+"Proxy)element).getID();");
addBody+= concat(tableManager+"...increase(elementID);")
addBody+= setPropertyExpression(ID,persCls , 'persCls ': 'elementID');
for each property p in properties(element)
    addBody+=setPropertyExpression(p,persCls , 'persCls ': 'elementID');

removeSig = "public Object remove(int index)";
removeBody = "'int elementID'';
removeBody+= retrievePropertyExpression('persCls 'ID,
persCls , 'persCls ': 'index ');
removeBody+= concat(tableManager+"...decrease(elementID);")
removeBody+= concat("'return new "+persCls+"Proxy(elementID);")

getSig = "public Object get(int index)";
getBody = "/same implementation as in removeBody
without notifying the TableManager";

sizeSig = "public int size()";
sizeBody = countPropertiesExpression(ID,persCls , 'persCls ': 'servID ');

addClass(collName);
```
addField(collName,"private int ID");
addConstructor(collName,consSig);
addMethodImplementation(collName,consSig,"ID = colId;”);

addMethodSignature(collName,addSig);
addMethodImplementation(collName,addSig,addBody);

addConstructor(collName,removeSig);
addMethodImplementation(collName,removeSig,removeBody);

addMethodSignature(collName,getSig);
addMethodImplementation(collName,getSig,getBody);

addMethodSignature(collName,sizeSig);
addMethodImplementation(collName,sizeSig,sizeBody);

for each factoryMethod f in factories(collection)
  redirectFactoryMethod(collection,collName,f);

Example: The execution result of:
createP2PCollectionClass(OutgoingFolder,MessageProxy,MessagesTableManger)
is detailed in Listing 8.20.

Listing 8.20: MessagesCollectionMapperImp persistent collection class implementation

```java
public class MessagesCollectionMapperImp implements List{

  private int ID;

  //DB related parameters - URL, connection etc…

  private MessagesCollectionMapperImp(int colID) {
      //connection initialization…
      …
      ID = colID;
  }

  public void add(int index, Object element) {
      String subject = ((MessageProxy)element).getSubject();
      String body = ((MessageProxy)element).getBody();
      int elementID = ((MessageProxy)element).getID();
      MessagesTableManager.getUniqueInstance().increase(elementID);
  }
}
```
8.1. DAL specific transformations

20 String query = "INSERT INTO MessagesP(subject, body, elementID, ID, index) VALUES(
21 "" + subject + ", " + body + ", " + elementID + ", " + ID + ", " + index + ");"
22 // executes query ...
23
24 public Object remove(int index) {
25 // decrease tblmgr and constructs a new proxy to be returned
26 int elementID;
27
28 String query = "SELECT * FROM MessagesP WHERE index = " + index + ";";
29 // execute query -> elementID = res.getInt("messageID");
30 return (new MessageProxy(elementID));
31 }
32
33 public Object get(int index) {
34 MessageProxy element = null;
35
36 String query = "SELECT * FROM MessagesP WHERE index = " + index + ";";
37 // executes query and constructs a new MessageProxy element...
38 return element;
39 }
40
41 public int size() {
42 String query = "SELECT count(*) FROM MessagesP WHERE servID = " + ID + ";";
43 // execute query -> int result ...
44 return result;
45 }
46 // Implements all other List interface methods...
47 }
48

8.1.20 IntroduceTableManager

Usage: Persistent To Persistent and Persistent To Multi Persistent

Motivation: This transformation is a composition of the following two DAL transformations:
CreateTableManager and AssignTableManager.

Arguments:
8.2 General purpose transformations

8.2.1 retrievePropertyExpression

Arguments:

1. propertyName property [required]
2. schemaElementName schemaElement [required]

Precondition:

- schemaElement is identified within the data-layer.
- Property with name 'property' exists in schemaElement.

Action:

```java
if (existTableManager(persCls)==false) 
  then 
    createTableManager(persCls);
if (isProxyAssigned(persCls.proxy, persCls.tableManager) == false) 
  then 
    assignTableManager(persCls.proxy, pers.tableManager);
```
8.2. General purpose transformations

**Intuitive Description:** This transformation constructs a platform specific expression that retrieves a property value from a schemaElement identified within the data layer.

**Example:** The execution result of:

```java
retrievePropertyExpression('subject', '[messages].[id=3]')
```

may produce the following SQL query expression corresponding to a Relational-Storage:

```sql
SELECT subject from messages where id=3;
```

Alternatively, the transformation may generate an instruction correlated with a J2EE environment as follows:

```java
home = (MessageHome) PortableRemoteObject.narrow(ctx.lookup("MessageEntity"), MessageEntityHome.class);
MessageEntity bean = home.findByPrimaryKey(3);
String subject = bean.getSubject();
```

---

### 8.2.2 setPropertyExpression

**Arguments:**

1. `propertyName` property [required]
2. `propertyValue` value [required]
3. `schemaElementName` schemaElement [required]

**Precondition:**

- `schemaElement` is identified within the data-layer.
- Property with name 'property' exists in `schemaElement`.

**Action:**

**Intuitive Description:** This transformation constructs a platform specific expression that adds a property with a given value to an existing `schemaElement` identified within the data layer.

**Example:** The execution result of:
8.2. General purpose transformations

setPropertyExpression('title','hello','[messages].[id=3]'))

may produce the following SQL query expression corresponding to a Relational-Storage:

```
INSERT INTO messages (title , id ) VALUES ( 'hello ',3);
```

Alternatively, the transformation may generate an instruction correlated with a J2EE environment as follows:

```
Message bean = home.create("hello","3");
```

8.2.3 updatePropertyExpression

Arguments:

1. propertyName property [required]

2. propertyValue value [required]

3. schemaElementName schemaElement [required]

Precondition:

- schemaElement is identified within the data-layer.
- Property with name 'property' exists in schemaElement.

Action:

Intuitive Description: This transformation constructs a platform specific expression that updates a property value to an existing schemaElement identified within the data layer.

Example: The execution result of:

updatePropertyExpression('title','goodbye','[messages].[id=3]'))

may produce the following SQL query expression corresponding to a Relational-Storage:

```
UPDATE messages SET title='goodbye' WHERE id=3;
```

Alternatively, the transformation may generate an instruction correlated with a J2EE environment as follows:
8.2. General purpose transformations

MessageEntity bean = home.findByPrimaryKey(3);
bean.setTitle("goodbye");

8.2.4 deleteFromStorageExpression

Arguments:

1. schemaElementName schemaElement [required]

Precondition:

- schemaElement is identified within the data-layer.

Action:

Intuitive Description: This transformation constructs a platform specific expression that removes an existing schemaElement from the data layer.

Example: The execution result of:

```
deleteFromStorageExpression('[messages].[id=3]')
```

may produce the following SQL query expression corresponding to a Relational-Storage:

```
DELETE FROM messages WHERE id=3;
```

Alternatively, the transformation may generate an instruction correlated with a J2EE environment as follows:

```
Message bean = home.create("hello","3");
...
bean.remove();
```

8.2.5 terminateConnectionExpression

Arguments:

1. schemaElementName schemaElement [required]

Precondition:
8.2. General purpose transformations

- schemaElement is identified within the data-layer.

**Action:**

**Intuitive Description:** This transformation constructs a platform specific expression that terminates the connection to the data-layer.

**Example:** The execution result of:

```
terminateConnectionExpression('[messagesDB]')
```

may produce the following java instruction to access the data-layer using JDBC:ODBC driver:

```
try {
    con.close();
}
catch(SQLException ex){
    ...
}
```

### 8.2.6 initializeConnectionExpression

**Arguments:**

1. schemaElementName schemaElement [required]

**Precondition:**

- schemaElement is identified within the data-layer.

**Action:**

**Intuitive Description:** This transformation constructs a platform specific expression that initializes a connection to the data-layer.

**Example:** The execution result of:

```
initializeConnectionExpression('[messagesDB]')
```

may produce the following java instruction to access the data-layer using JDBC:ODBC driver:
try {
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
} catch (java.lang.ClassNotFoundException e) {
    ...
}
try {
    con = DriverManager.getConnection("jdbc:odbc:MessagesDB");
} catch (SQLException ex) {
    ...
}

8.2.7 replaceContructorsWithFactoryMethod

Arguments:

1. ClassName cn [required]

2. ConstructorSignature cSig [required]

Precondition:

- Class with name "cn" exists.
- Class with name "cn" has a constructor with signature cSig.
- Class with name "cn" doesn’t have a method named "create" with the same signature as cSig.

Action:

Intuitive Description: Applies the refactoring Replace Constructor with Factory Method [10]. For the constructor method argument given as 'cSig', a new factory method with the same signature is generated in the given class name 'cn'.

Example: Given the code in Listing 8.21, the execution result of:

replaceConstructorWithFactoryMethod(MyClass, MyClass(int num))

is detailed in Listing 8.22.
8.2. General purpose transformations

Listing 8.21: replaceContractorsWithFactoryMethod - Before transformation

```java
public class MyClass{
    public MyClass(int num){
        ...
    }
}
```

Listing 8.22: replaceContractorsWithFactoryMethod - After transformation

```java
public class MyClass{
    public static MyClass create(int num){ //Factory method
        return new MyClass(num);
    }
    private MyClass(int num){
        ...
    }
}
```

Precise Description:

let: factSig = concat("public","static",cn,"create","parameters(cn, cSig)",")")

    factBody = "return new "+cn+"("+parameters(cn,cSig)+")";

    newInvokation = concat(cn,".create","parameters(cn, cSig)",")")

        addMethodSignature(cn, factSig);

        addMethodImplementation(cn, factSig, factBody);

        setAccessSpecifier(cn, cSig, "private");

        For each element in invok(cn, cSig):
            replace element by newInvokation;

8.2.8 extractSubClass

Arguments:

1. className superClassName [required]

2. className subClassName [required]

Precondition:
1. Class with name 'superClassName' exists.

2. Class with name 'subClassName' doesn’t exist.

**Action:**

**Intuitive Description:** Applies the refactoring *Extract Subclass* [10]. All instance methods and fields of the class 'superClassName' are pushed down to a new sub-class named 'subClassName'. If the super-class has also abstract-methods, the same method signature is added to the sub-class for each abstract-method. Pushing down methods leaves only abstract methods signature in the base-class.

**Example:** Given the code in Listing 8.23, the execution result of:

`extractSubClass(MyClass,NewClass)`

is detailed in Listing 8.24.

**Listing 8.23: extractSubClass - Before transformation**

```java
public class MyClass{
    private String name;
    public MyClass(int num){
        ...
    }
}
```

**Listing 8.24: extractSubClass - After transformation**

```java
public class MyClass{
    public abstract MyClass(int num){}
}

public class NewClass{
    private String name;
    public MyClass(int num){
        ...
    }
}
```
8.2. General purpose transformations

Precise Description:

\[
\begin{align*}
&\text{addClass(subClassName);} \\
&\text{addSubTypeLink(subClassName,superClassName);} \\
&\text{For each element } f \text{ in fields(superClassName)} \\
&\text{addField(subClassName,f);} \\
&\text{removeField(superClassName,f);} \\
&\text{For each element } m \text{ in methods(superClassName)} \\
&\quad \text{addMethodSignature(subClassName,SignatureOf(m));} \\
&\quad \text{if (getMethodStereotype(m)=='abstract')} \\
&\quad \quad \text{addMethodImplementation(subClassName,SignatureOf(m),bodyOf(m));} \\
&\quad \text{removeMethodImplementation(superClassName,SignatureOf(m),bodyOf(m));} \\
&\quad \text{addMethodStereotype(m,'abstract');}
\end{align*}
\]

8.2.9 \texttt{addConstructor}

Arguments:

1. className cls [required]
2. methodSignature mSig [required]

Precondition:

1. Class with name 'cls' exists.

Action:

\textbf{Intuitive Description:} A new method signature is added to the given class 'cls' only if the name of the method equals to the name of the class 'cls'.

\textbf{Example:} Given the code in Listing 8.25, the execution result of:

\texttt{addConstructor(MyClass,"public MyClass(int num)"})

is detailed in Listing 8.26.
8.2. General purpose transformations

Listing 8.25: addConstructor - Before transformation

```java
public class MyClass {
    ...
}
```

Listing 8.26: addConstructor - After transformation

```java
public class MyClass {
    public MyClass(int num) {
    }
}
```

Precise Description:

```java
if (cls equals mSig)
    addMethodSignature(cls, mSig);
```

8.2.10 redirectFactoryMethod

Arguments:

1. className superCls [required]
2. className subCls [required]
3. methodSignature constructor [required]

Precondition:

Action:

Intuitive Description: Changes the instance type returned by a FactoryMethod with same arguments list as in the given 'constructor', to be the type of the given class 'subClass'.

Example: Given the code in Listing 8.27, the execution result of:

```java
redirectFactoryMethod(MyClass, OtherClass, (int num))
```

is detailed in Listing 8.28.
### 8.3. Primitive conditional transformations

#### Listing 8.27: redirectFactoryMethod - Before transformation

```java
public class MyClass {
    public static void create(int num) {
        return new MyClass(num);
    }
}
```

#### Listing 8.28: redirectFactoryMethod - After transformation

```java
public class MyClass {
    public static void create(int num) {
        return new OtherClass(num);
    }
}
```

**Precise Description:**

```
let: argesList = constructor.getArguments();
newFactImp = concat("return new " + subCls + "(" + getParameters(constructor) + ");");

For each fMethod in factoryMethods(superCls)
    if arguments(fMethod) == arguments(constructor)
        addMethodImplementation(superCls, fMethod, newFactImp);
```

8.3 **Primitive conditional transformations**

**addClass(ClassName)**

**precondition:** no class in that name.

**addField(ClassName, FieldSignature)**

**precondition:** class exists and has no field with that name.
8.3. Primitive conditional transformations

**removeField(ClassName, FieldSignature)**

**precondition:** class exists and has a field with that name.

**addMethodSignature(ClassName, MethodSignature)**

**precondition:** class exists and has no method with that name.

**addMethodimplementation(ClassName, MethodSignature, Expression)**

**precondition:** expression should be a legal Java statement/sequence of statements/expression. i.e., a legal method body, following the Java syntactical specification.

**removeMethodimplementation(ClassName, MethodSignature)**

**precondition:** none.

**addClassStereotype(ClassName, Stereotype)**

**precondition:** assuming a finite type Stereotype that includes a finite number of strings.

**removeClassStereotype(ClassName, Stereotype)**

**precondition:** assuming a finite type Stereotype that includes a finite number of strings.

**addSubTypeLink(Classname c1, Classname c2)**

**precondition:** classes exist, c2 has no super.

**removeSubLink(Classname c1, Classname c2)**
precondition: classes exist, c2 has c1 as its super.

setAccessSpecifier(ClassName,MethodSignature,Specifier)

precondition: Specifier public, private, protected, package
Chapter 9

Appendix B - Smart Unloading Mechanism

As mentioned in the \textit{addDelegatedMethod} transformation detailed in the previous chapter, this chapter illustrates an alternate implementation for the \textit{Proxy} class that uses a different approach for loading and unloading persistent objects. In this example, the \textit{Proxy} class uses an independent scheduling process (implemented as a java thread) for loading and unloading the fullweight persistent object representative to and from memory. This mechanism is an optimization which reduces storage access rate and therefore improves the performance of the core PDM pattern implementation.

Figure 9.1 illustrates an enhanced Proxy class that supports scheduling. Figure 9.2 illustrates an unloader thread that is responsible for releasing the persistent object from being memory resident whenever it is not required.

Listing 9.1: Improved Proxy with an unloading mechanism

```
public class MessageProxy extends Message {
    private int ID;
    private MessageImp message;
    private MessageDataMapper dataMapper;
    private long lastAccessTime;
```
private UnLoader unLoader;

protected MessageProxy(String title) {
dataMapper = MessageDataMapper.getUniqueInstance();
message = new MessageImp(title);
lastAccessTime = 0;
// Define an UnLoader thread above this proxy
// and set checking interval to the in-memory object each 100 milliseconds
unLoader = new UnLoader(this, 100);
// Updates first Access Time
lastAccessTime = (new Date()).getTime();
// Start the unLoader thread operation here
unLoader.start();
ID = dataMapper.store(message);
}

protected void finalize() throws Throwable {
dataMapper.delete(ID);
}

private MessageImp load() {
    return dataMapper.load(ID);
}

private void update(MessageImp mes) {
dataMapper.update(mes, ID);
}

public void unload() {
    message = null;
}

public long getLastAccessTime() {
    return lastAccessTime;
}

...
active = true;
System.out.println("UnLoader\ created.");
}

public void setActive()
active = true;
System.out.println("UnLoader\ reactivated.");
}

public void run()
while(true)
hold(checkInterval);
if (active)
long lastAccess;
lastAccess = proxy.getLastAccessTime();
long now = (new Date()).getTime();
if ((now-lastAccess) >= checkInterval)

proxy.unload();
System.out.println("Mapper unloaded its in-memory instance.");
active = false;
System.out.println("UnLoader\ disactivated.");

private void hold(long ms)
try
sleep(ms);