Enterprise Application Design Patterns: Improved and Applied

Stuart Thiel

A Thesis
in
The Department
of
Computer Science
and
Software Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

January 2010

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: Stuart Thiel

Entitled: Enterprise Application Design Patterns: Improved and Applied

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.
Signed by the final examining committee:

_____Chair

Dr. Nematollaah Shiri

_____Examiner

Dr. Greg Butler

_____Examiner

Dr. Yuhong Yan

_____Supervisor

Dr. Patrice Chalin

Approved by _____

Chair of Department or Graduate Program Director

_____

Dr. Robin Drew, Dean

Faculty of Engineering and Computer Science

Date _____

Abstract

Enterprise Application Design Patterns: Improved and Applied

Stuart Thiel

Providing developers with proper tools is of ever increasing importance as software integrates itself further into all aspects of our lives. Aside from conventional hardware and software tools, architectural and design patterns have been identified over the years as a means to communicate knowledge of known problems and their solutions. In this thesis, we present several refinements and additions to these patterns, building primarily on Martin Fowler's *Patterns of Enterprise Application Architecture* (2003). We present a practical implementation approach to using these patterns and discuss a framework that we have developed to aid practitioners in following this methodology. We also incorporate several of Martin Fowler's existing patterns into an iterative design example to better demonstrate progressively improving combinations of their use in existing systems.

Acknowledgements

I have never met Martin Fowler, and many people I know believe that I dislike him. Nothing could be further from the truth. His work in *Patterns of Enterprise Application Architecture* has inspired me for nearly a decade. While I have continually strived to surpass his work, I am truly standing on the shoulders of a giant.

I would also like to thank my supervisor and friend, Dr. Patrice Chalin. Without someone of his caliber to discuss these ideas with, I do not believe I could have achieved as much so quickly. His intelligence and precision have encouraged me to always be more. I have also greatly appreciated teaching with him, and being taught by him. Virtually every ounce of diplomacy I have, I probably owe to him.

I would like to thank the DSRG over the last few years for tirelessly helping me with the various incarnations of my thesis, through our reading and writing workshops where we reviewed each others current papers, or through long discussions around scrap paper, and sometimes pints. Daniel Sinnig, Perry James, Rajiv Abraham, Stephen Barret, Asif Dogar, George Karabotsos and Kianoush Torkzadeh, you have had a great impact on my life and on this thesis.

I would lastly like to thank my friends and family who have continually supported me and encouraged me to hurry up and finish. I would like to particularly thank my wife Karen Bennett for her patience with me, and my newborn daughter Kathryn who has graciously let me sleep well most nights, and who has bubbled and chirped happily next to me while I worked on this thesis. I would also like to thank Finn Upham, Jeremy Upham, Susan Upham (yes, the entire family independently), My-An Nguyen, Larry Thiel (my father) and David Reisch, who have each contributed a final review that will undoubtedly have greatly improved the overall quality of this thesis.

Table of Contents

1 Introduction 1
1.1 Problem 1
1.2 Contributions 2
1.3 Scope 2
2 Background 3
2.1 Architectural Styles: Layered and Client-Server 3
2.2 A layered approach 3
2.3 Client-Server 4
3 WEA Design Patterns in Practice: A Tutorial and Critical Assessment 5
3.1 Introduction to the BuddyAge Application 5
3.2 Iteration Style 6
3.3 Iteration 1: Do-It-All TRANSACTION SCRIPTS 6
3.3.1 Pattern Description 6
3.3.2 Pattern Usage 7
3.3.3 Concerns 7
3.4 Iteration 2: Isolating Technical Services with ROW DATA GATEWAY (RDG) 7
3.4.1 Pattern Description 8
3.4.2 Pattern Usage 8
3.4.3 Concerns 9
3.5 Iteration 3: Isolating Presentation with TEMPLATE VIEW and VIEW HELPER 9
3.5.1 Pattern Description 9
3.5.2 Pattern Usage 10
3.5.3 Concerns 11
3.6 Iteration 4: Data Integrity and an Isolated Domain Logic with OPTIMISTIC OFFLINE LOCK, PAGE CONTROLLER and IDENTITY FIELDS 11
3.6.1 Pattern Description 13
3.6.2 Pattern Usage 15
3.6.3 Concerns 16
3.7 Iteration 5: Refined Access to the Data Source with DATA MAPPER, TABLE DATA GATEWAY and DOMAIN MODEL 17
3.7.1 Pattern Description 18
3.7.2 Pattern Usage 19
3.7.3 Concerns 19
3.8 Iteration 6: An Organized Approach to the Application Layer Using the FRONT CONTROLLER Pattern 20
3.8.1 Pattern Description 20
3.8.2 Pattern Usage 21
3.8.3 Concerns 21
3.9 Iteration 7: Managing In-Memory Data with LAZY LOAD (via VIRTUAL PROXY) and IDENTITY MAP 21
3.9.1 Pattern Description 23
3.9.2 Pattern Usage 23
3.9.3 Concerns 24
3.10 Iteration 8: Accommodating a Complex Domain with UNIT OF WORK (UOW) and DEPENDENT MAPPING 24
3.10.1 Pattern Description 26
3.10.2 Pattern Usage 27
3.10.3 Concerns 27
4 WEA Design Patterns Revisited 28
4.1 DOMAIN OBJECT 28

4.1.1 Context 28
4.1.2 Problem 28
4.1.3 Solution 28
4.1.4 Related work and contribution 29
4.2 Front Controller, Dispatchers and Commands 30
4.2.1 Context 30
4.2.2 Problem 30
4.2.3 Solution 30
4.2.4 Related work and contribution 32
4.3 Lazy Load: Domain Object Proxy and List Proxy 32
4.3.1 Context 32
4.3.2 Problem 33
4.3.3 Solution 33
4.3.4 Related work and contribution 34
4.4 Identity Map 34
4.4.1 Context 34
4.4.2 Problem 35
4.4.3 Solution 35
4.4.4 Related work and contribution 36
4.5 Input Mapper and Output Mapper Patterns 36
4.5.1 Context 36
4.5.2 Problem 36
4.5.3 Solution 36
4.5.4 Related work and contribution 41
4.6 Table Data Gateway (TDG) and Finder 42
4.6.1 Context 42
4.6.2 Problem 42
4.6.3 Solution 43
4.6.4 Related work and contribution 45
5 Applied and Improved Design: The SoenEA Framework and its Use 46
5.1 SoenEA, our WEA Framework 46
5.2 SoenEA Patterns 47
5.2.1 Domain Objects 48
5.2.2 GenericOutputMapper 50
5.2.3 UoW and IdentityMap 52
5.2.4 ListProxy 54
5.2.5 Dispatcher and DomainCommand 56
5.2.6 InputMapper 57
5.2.7 TDG/Finder 58
5.3 SoeanEA DITCs 63
5.3.1 User 63
5.3.2 Role 63
5.3.3 DispatcherServlet and Servlet 64
5.3.4 Helper 64
5.4 SoenEA Utilities 64
5.4.1 DispatcherFactory 65
5.4.2 UniqueIdFactory, UniqueIdTDG/Finder 65
5.4.3 MetaDomainObject 65
5.4.4 MapperFactory and MetaMapper 65
5.4.5 DBRegistry, ConnectionFactorys and Connections 66
5.4.6 ApplicationAuthorization 66
5.4.7 ThreadLocalTracker 66
5.4.8 Exceptions 67

5.5 SoenEA Test Suite 67
6 Professional Software Development Using SoenEA 68
7 Conclusion 69
8 References 71
9 Appendix 73

List of Figures

Figure 2-1 Three layer style of a WEA....3
Figure 2-2 A more accurate distribution of client-server over the three layer architecture....4
Figure 3-1 Navigating BuddyAge application pages....5
Figure 3-2 Sample “Browse People” web page....5
Figure 3-3 Sample “View Person” web page....5
Figure 3-4 Class diagram of TRANSACTION SCRIPTS spanning all three layers....6
Figure 3-5 Isolating data source access in a ROW DATA GATEWAY....8
Figure 3-6 Class diagram showing the addition of TEMPLATE VIEWs and a VIEW HELPER....10
Figure 3-7 Warning User 2 that someone else has already updated Bob’s age....12
Figure 3-8 A class diagram showing PAGE CONTROLLERS, IDENTITY FIELD and OPTIMISTIC OFFLINE LOCK....13
Figure 3-9 Two transactions rendered sequential....14
Figure 3-10 Deadlock resolution on conflicting transactions....14
Figure 3-11 Single table scan approach....16
Figure 3-12 Many applications with many databases....16
Figure 3-13 Person, PersonMapper and PersonTDG replace PersonHelper and PersonRDG....18
Figure 3-14 Using a FRONT CONTROLLER and COMMANDS to replace PAGE CONTROLLERS....20
Figure 3-15 Viewing a Person and their buddy....22
Figure 3-16 Alice is Bob's buddy. Bob is Alice's buddy....22
Figure 3-17 Adding a VIRTUAL PROXY and IDENTITY MAP....22
Figure 3-18 PersonProxy code for getting a real Person and illustrating delegation....23
Figure 3-19 UOW and a relationship between PERSON and PHONENUMBER....25
Figure 3-20 PersonMapper methods....26
Figure 4-1 An implementation of Person using the Domain Object Pattern....29
Figure 4-2 A simple Login Dispatcher using SoenEA....31
Figure 4-3 A simple Login Command using SoenEA....31
Figure 4-4 Separation between Dispatcher, View, Command and Domain Object Patterns....32
Figure 4-5 An Object Diagram showing two instances related in both directions by the role buddy....33
Figure 4-6 An example OutputMapper delete method....37
Figure 4-7 Output Mappers store domain logic regarding object relations....39
Figure 4-8 example of tables for a Person described in Figure 4-7....39
Figure 4-9 Two tables, representing a one-to-one relationship....40
Figure 4-10 The concrete TDG for the Person Domain Object....40
Figure 4-11 inserting a person....41
Figure 4-12 updating a Person....41
Figure 4-13 [XKCD, “Exploits of a Mom”, http://xkcd.com/327/]....43
Figure 4-14 How DOs, I/O Mappers, TDGs and Finders relate....43
Figure 4-15 update in a TDG....44
Figure 5-1 SoenEA general usage diagram....47
Figure 5-2 Domain Objects....48
Figure 5-3 GroupFactory Methods....49
Figure 5-4Creating Domain Objects....50
Figure 5-5 GenericOutputMapper....50
Figure 5-6 Creating a GENERICOUTPUTMAPPER....51
Figure 5-7 GroupOutputMapper Methods....52
Figure 5-8 UoW backs IdentityMap....52
Figure 5-9 Sample code initializing the UoW with DomainObjects and OutputMappers....53
Figure 5-10 Code demonstrating the use of the UoW and IdentityMap in an InputMapper....54
Figure 5-11 SetProxy and ListProxy....55
Figure 5-12 MembershipListProxy....55
Figure 5-13 sample code from GroupMembershipInputMapper....56
Figure 5-14 Dispatcher and DomainCommand with support classes....56
Figure 5-15 Detailed Class diagram of GroupMembershipInputMapper....57

Figure 5-16 the getGroupMembership method....58
Figure 5-17 Class diagram of Group TDG/Finder....59
Figure 5-18 GroupTDG's update method....60
Figure 5-19 GroupFinder's find identity find method....61
Figure 5-20 Summary of Domain Objects seen in this section....62
Figure 5-21 A simple class diagram showing the domain layer relating to the technical services layer....63
Figure 5-22 MyResources.properties and Access.xml....64
Figure 5-23 Use of a UniqueIdFactory....65
Figure 5-24 UoW using MapperFactory and MetaMapper....66
Figure 9-1 A recommended directory structure....74
Figure 9-2 FrontController....78
Figure 9-3 LoginDispatcher....78
Figure 9-4 LogoutDispatcher....79
Figure 9-5 LoginCommand....79
Figure 9-6 IGroup....80
Figure 9-7 Group....80
Figure 9-8 GroupProxy....81
Figure 9-9 GroupFactory....82
Figure 9-10 GroupInputMapper....83
Figure 9-11 GroupOutputMapper....84
Figure 9-12 GroupTDG....85
Figure 9-13 GroupFinder....86
Figure 9-14 IGroupMembership....87
Figure 9-15 GroupMembership....88
Figure 9-16 GroupMembershipProxy....89
Figure 9-17 GroupMembershipFactory....90
Figure 9-18 MembershipStatus....90
Figure 9-19 MembershipListProxy....90
Figure 9-20 GroupMembershipInputMapper....91
Figure 9-21 GroupMembershipOutputMapper....92
Figure 9-22 GroupMembershipTDG....93
Figure 9-23 GroupMembershipFinder....94
Figure 9-24 MemberInputMapper....95
Figure 9-25 AdminRole....95
Figure 9-26 RegisteredRole....96
Figure 9-27 RoleIds....96
Figure 9-28 DatabaseSetup....99
Figure 9-29 Access.xml....99
Figure 9-30 MyResources.properties....100

List of Terms/Acronyms

Term/Acronym	Definition
ACID	Atomicity, Consistency, Isolation, Durability. A set of properties, which taken together allow data sources to be accessed in a convenient and safe fashion. Without ACID compliant transactional resources, Web Enterprise Applications would be very difficult to work with reliably.
CRUD	CRUD stands for Create, Retrieve, Update, Delete. These general operations represent the majority of activity that happens with Object-Oriented data. A CRUD approach is one where these operations are applied systematically across all or most data in a system.
Enterprise Applications (EAs) / Web Enterprise Applications (WEAs)	Enterprise Applications (EAs) are generally understood to be on-demand, user-interaction based applications that are meant to be accessed by multiple users, usually from the same organization. Web-based Enterprise Applications (WEAs) imply EAs made available through the Internet. These applications (EAs and WEAs) generally use databases for persistent storage.
GoF	Gang of Four. Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides authored *Design Patterns: Elements of Reusable Object-Oriented Software*, a book that describes 23 well know design patterns. The authors are referred to as the GoF, and these patterns are referred to as the GoF patterns.
GRASP	General Responsibility Assignment Software Patterns. GRASP patterns suggest solutions to common problems of responsibility assignment. These patterns represent basic principles of Object-Oriented software design. [Larman 2004]
GUI	Graphical User Interface. Often seen as the more general term UI (User Interface)
High Cohesion	A GRASP pattern. A module with high cohesion has only closely related responsibilities; applying High Cohesion as an evaluative pattern encourages modules that have more closely related responsibilities.
IDE	Integrated Development Environment. Usually a programming environment that incorporates a compiler, debugger and standard output for run code. The modern IDE includes integration with revision control systems, complex auto-complete features and many other advanced features to make development easier.
Java Server Pages (JSP)	Often identified separately from Servlets, JSPs are the basic form of the template language provided when working with Servlets. JSPs are transformed into java for mini Servlets; this java is then compiled as needed. JSPs also have more complex services, but the main purpose is to provide a simple template-like language for the easy generation of responses to user requests while still allowing access to code. In the most basic form, the syntax is quite similar to that of Active Server Pages (ASP) and PHP.
LOC	Lines Of Code. A readily acquired metric for measuring software. A related term is SLOC (Source Lines Of Code) that counts only those lines that are not white space or comments.
Low Coupling	A GRASP pattern. A module with low coupling relies on few other modules; applying Low Coupling as an evaluative pattern discourages modules that rely on many other modules.

RDG	A Row Data Gateway is a simple data access pattern described in [Fowler 2004], merging the responsibility of retrieving data from a data source with its in memory representation.
Servlet	A Servlet is a Java application running within a web application environment on a server. The technology that enables Servlets provides access to web request data and a means to respond in kind.
Servlet Container	A Servlet Container is the technology that enables and wraps a Servlet. Several free and commercial Servlet Containers are available today. While the manner in which a Servlet interacts with a Servlet Container is standardized, the particulars of the implementation of these Containers can vary, highlighting different intended uses or development philosophies. Most of the time, these details should not directly affect development, but occasionally there are surprises.
Servlet Context	A Servlet Context is the means of identifying a section of a Servlet Container that holds one or more Servlets. While one Servlet is instantiated for all requests to that Servlet, it is possible to load the same classes as a separate Servlet within the same Servlet Context, sharing static variables and other global information across all Servlets within that Servlet Context.
SoenEA	Software Engineering (SOEN) Enterprise Application (EA). This is the name of a framework that we have developed in conjunction with this thesis to aid in the rapid development of dependable Web Enterprise Applications.
Tomcat	Tomcat is the Servlet Container that Dr. Chalin and Stuart Thiel have used on their own and in the Software Design/Architecture courses at Concordia University, in various incarnations, for nearly a decade. All our Servlet-based web applications and testing environments have been developed with versions of Tomcat from 4.1 through 6.x. Tomcat is developed under the Apache Software License and is freely available. It was previously part of the Apache Jakarta project, but now is a project in its own right.
UML	Unified Modeling Language. Text, lines and boxes as a means of communicating in Object-Oriented analysis and design.

1 Introduction

Enterprise Applications (EAs) are generally understood to be on-demand, user-interaction based applications that are meant to be accessed by multiple users, usually from the same organization. Web-based Enterprise Applications (WEAs) imply EAs made available through the Internet. These applications (EAs and WEAs) generally use databases for persistent storage. E-commerce sites (such as Amazon [AMAZON] and eBay [EBAY]), banking sites, webmail, online casinos and search engines are some of the many examples of WEAs. Since 2000, US retail e-commerce sales have steadily increased their percentage of the overall retail sector, achieving an estimated 3.6%[1] in the second quarter of 2009 [USCENSUS], highlighting the emerging importance of WEAs in the market.

Keeping up with the advancement of WEAs requires more than just single developers. Large teams comprising many roles are now quite normal. With the increased manpower and separation of roles comes an increased need for communication and accountability. More standardized tools, languages and approaches need to be developed and adopted to ensure reliable results. One of the keys to improving the language of communication used in this field is the use of design patterns. Similar to Larman we take a software design pattern to be a named problem/solution pair. Larman, consistent with most definitions of "design pattern", also states that a pattern is a *well-known* problem/solution pair [Larman 2004].

Martin Fowler introduces 51 architectural patterns in his book, "Patterns of Enterprise Application Architecture" [Fowler 2003]. The book describes how developers can use these patterns to understand existing Enterprise Applications and to better write new ones. Fowler's work provides a stable basis to further introduce good practices to Software Engineers.

1.1 Problem

While there are many areas that warrant improvement in the field of EAs in general and WEAs in particular, we have focused on areas that are important for learning and comprehension. In particular, we have sought to add examples, improve guidance on usage and provide a clearer separation between theory and practice. More specifically, we have identified the following problems:

- High level patterns lack comprehensive examples.
- Guidance regarding the use of interrelated WEA patterns is sparse.
- The separation between theory and implementation is ambiguous.

While current design patterns contribute to thinking about design, help identify design patterns in practice and are a means to communicate those patterns to others, many of the higher level design patterns do not provide much prescriptive guidance for developers, particularly guidance needed by those without much experience. For example, a broader architectural pattern, such as LAZY LOAD [Fowler 2003] addresses a fairly specific problem, but one needs additional patterns to describe the various avenues to approach a solution. Guidance as to when to use which sub-pattern (GHOST or VIRTUAL PROXY) is minimal..

Many of the higher level patterns are paired with others with which they work well (e.g. DOMAIN MODELS with MAPPERS; LAZY LOAD with VIRTUAL PROXY). In [Fowler 2003], code is often provided showing implementations of these patterns in isolation. It would be more helpful to have guidelines showing the implications of mixing many of these patterns in a standard configuration. Such examples would more readily guide developers in the use of the patterns and their potentially complex interactions.

In [Fowler 2003], the theory of each pattern is discussed abstractly while subsequent reasoning and examples are often given in a practical manner, albeit one that is narrow and geared strongly towards implementation. Increased separation between the theory and implementation and a closer examination of the theory would give developers better tools when deciding between patterns.

These problems highlight some of Fowler's admittedly inherent shortcomings in his book, as it was intended as an introductory resource and represents only a snapshot of his evolving understanding. As such, in critiquing his work we are building on his already formidable starting point.

[1] $32.4 billion of $906 billion total retail

1.2 Contributions

The main contributions of this thesis address the previously mentioned problems through

- The suggestion of additional patterns.
- Improved definitions for some existing patterns.
- More concrete guidelines regarding the application of patterns.
- In particular, we offer guidelines to address integrating compatible patterns.
- The introduction of a framework, SoenEA, designed to facilitate implementing the theory described herein

In addition to expanding on existing patterns and providing supporting implementation, as a secondary contribution we also

- Demonstrate how a subset of Fowler's patterns can work together in gradually more complex combinations, iteratively achieving better design.
- Provide a sample implementation integrating our recommended combination of patterns.

1.3 Scope

Using Fowler's suggested patterns and code [Fowler 2003], a simple application will be put together. Using this example, Fowler's patterns will be briefly explained. Given a fair understanding of what Fowler has provided, the changes and additions to these patterns will be examined. Differences and divergence from Fowler's patterns will be highlighted. Once the theory has been covered, a practical examination of its application using the SoenEA framework will be given, referencing examples from the sample application that is available online in source form.

It is assumed that the reader is familiar with the patterns and theory described in [Fowler 2003], [Larman 2004]'s GRASP patterns and basic architectural styles, particularly those popularly applied to WEAs. In addition, the reader should be familiar with UML. A summary of some basic Software Engineering concepts will be provided in Chapter 2.

2 Background

In this chapter, the concept of architectural styles—the layered and client-server styles in particular—are reviewed to provide a basis for the rest of this thesis. A common layered scheme will also be discussed and used to help organize design patterns.

2.1 Architectural Styles: Layered and Client-Server

Though there are many definitions, we see an architectural style is a general way of thinking about and approaching a problem at a high level. Architectural styles are often described in terms of their components and connectors, as well as the rules for their interaction. Like design patterns, architectural styles are patterns that have been found to recur in proven systems.

In the *layered style*, layers are the components. Protocols that dictate how layers, and modules within layers may depend on each other define the connectors this style. In what we describe as a "pure" *layered style*–in that this variation follows the style guidelines suggested in [SG96] exactly–these protocols state that only adjacent layers may communicate; layers below provide services to the layer immediately above and layers below are oblivious to the layers above [SG96].

In the *client-server style*, components are clients and servers which are separated across a network. The connectors for this style are requests made over network links, which come only from the clients to the servers, and the subsequent responses.

One could, for example, have a client-server style system where the server side makes use of the layered style and the client side might also make use of the layered style. In this way a system's architecture will be described by potentially many styles, just as a software design will make use of multiple design patterns.

2.2 A layered approach

Web-based Enterprise Application (WEA) development is often done using a three layered architecture. [Larman 2004] and [Fowler 2003] both provide examples of such three layer organizations, and we combine their approaches to provide a similar breakdown shown in Figure 2-1, and described in the rest of this Section.

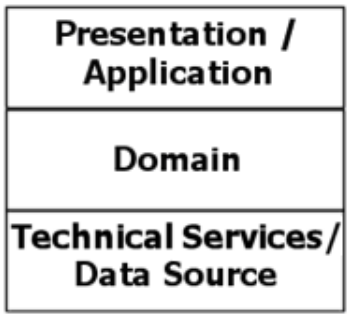


Figure 2-1 Three layer style of a WEA

In keeping with a pure *layered style* (this will become apparent in Chapter 3), we merge the Application and Presentation layers, both described in [Larman 2004], which differs from Larman's approach of keeping the Application layer separate if it is used at all. Application-specific elements are those elements that have more to do with the type of the application, (i.e. web application, desktop application, applet, etc.) than with the specific use of the application (i.e. selling computers, guiding a user on a dragon slaying experience, etc.) Presentation-specific elements would be those elements that a user would interact with visually, such as windows. These Presentation and Application—specific elements are placed in this uppermost layer.

The Domain layer contains the logic and entities which describes the area of concern of an application. For example, an application that acts as a digital rolodex could contain classes describing people, the Domain entities of the application; the application's behavior might include adding, removing and updating entries on people, which constitutes the Domain logic of the application. The Domain layer contains the logic which defines interaction between Domain entities (e.g. Person). An e-commerce site might contain different promotional billing strategies, and this could also be programmatically captured in this layer.

[Larman 2004] and [Fowler 2003], differ slightly in the use of terms for the bottom-most layer, "Technical Services" and "Data Source" respectively. As the "Technical Services" layer subsumes the "Data Source" layer, what [Larman 2004] describes in a "Persistence" package within "Technical Services", we generally use the term "Technical Services". The Technical Services layer will contain adapters to third party systems (e.g. tax calculators, shipping calculators) as well as mechanisms for communicating with data sources. A main purpose of this layer is to hide technical details that are specific to external systems—such as libraries and utilities—from the layers above. The technical details that a developer often wants to take for granted when thinking abstractly are often found here.

2.3 Client-Server

WEAs are client-server applications. Making a WEA means implementing the server side of the application as well as what will run on client machines (usually in a client's web browser). This separation does not coincide cleanly with the boundaries of the three layers shown in **Error! Reference source not found.**. A common assumption is that all presentation components should exist on the client side, but this does not take into account server-side decisions about presentation or security.

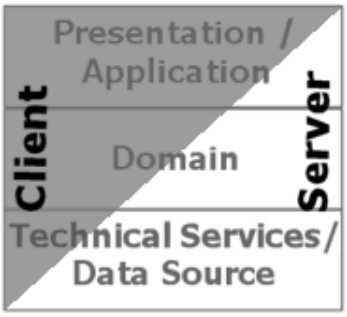


Figure 2-2 A more accurate distribution of client-server over the three layer architecture

Figure 2-2 gives an idea of what is on the client side vs. on the server side. The breakdown of client-side to server-side implementation is not clear-cut. The trend is that the Application / Presentation layer is usually more represented on the client side, and the Service layer is much less so. The Domain layer tends to be represented more evenly, usually leaning towards the server side.

While this thesis will focus on the server-side aspects of WEAs, it is important to understand the client-side aspect as well. For example, concerns that might be considered part of the Domain layer, such as ensuring that input from the user is valid, are often run on the client side. While client-side validation can be easily circumvented by someone who knows their browser[2], it does protect users of the application's front-end from tying up the system with multiple erroneous network requests due to small mistakes (e.g. formatting their postal code wrong, or missing a digit from a phone number). The tangible benefit for the end-user is responsiveness. Martin Fowler defines responsiveness as: "… how quickly the system acknowledges a request as opposed to processing it." [Fowler 2003, page 7] Processing of the request does not start until the client-side checking is done.

[2] As the validation is done on the client side, this validation is necessarily under the control of the client. Firefox, for example, has a plug-in called FireBug which can be used to change any part of a web page on the client's side, including JavaScript variables and cookies.

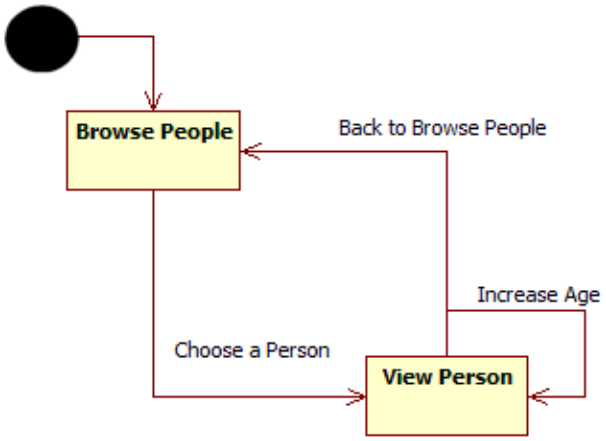


Figure 3-1 Navigating BuddyAge application pages

3 WEA Design Patterns in Practice: A Tutorial and Critical Assessment

In this chapter we provide an example that is simple, yet rich enough to illustrate the state of the art of Enterprise Application (EA) patterns. The chosen example is a buddy list with age information which we will call the BuddyAge application. The chapter is broken down into a series of iterations, each describing a version of BuddyAge.

The goal of reading through each iteration in this chapter is to develop a familiarity with Fowler's patterns (and one from Sun [Alur 2001]), on which the theory in this thesis is built. We have contributed a progression through the patterns that we feel helps in their practical understanding. We have also tried to describe how these patterns relate, beyond what is offered in Fowler's book [Fowler 2003].

3.1 Introduction to the BuddyAge Application

Our BuddyAge application allows a user to choose a name from a list of "buddies" and then see the chosen person's age. When viewing a person's age, it is also possible to increment that age (as one might do on a person's birthday).

Based on Figure 3-1 we can break down the application's functionality into the following initial feature list:

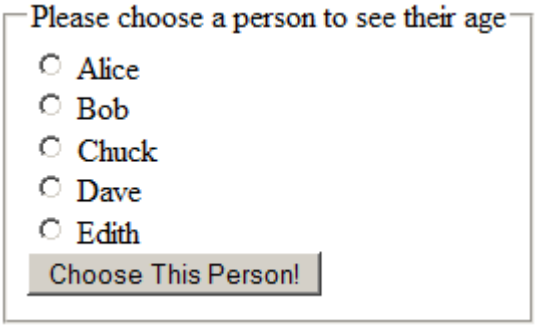


Figure 3-2 Sample "Browse People" web page

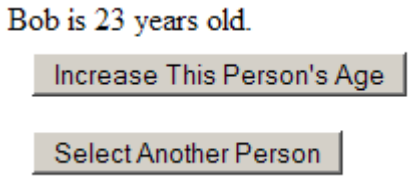


Figure 3-3 Sample "View Person" web page

- Browse People
- View Person
- Increase Age

In a session using this application, one might browse several listed names (Figure 3-2), choose a person and then view that person's age (Figure 3-3). When viewing a person, the system allows the user to return to the list of buddies or choose to increase the age of the person shown.

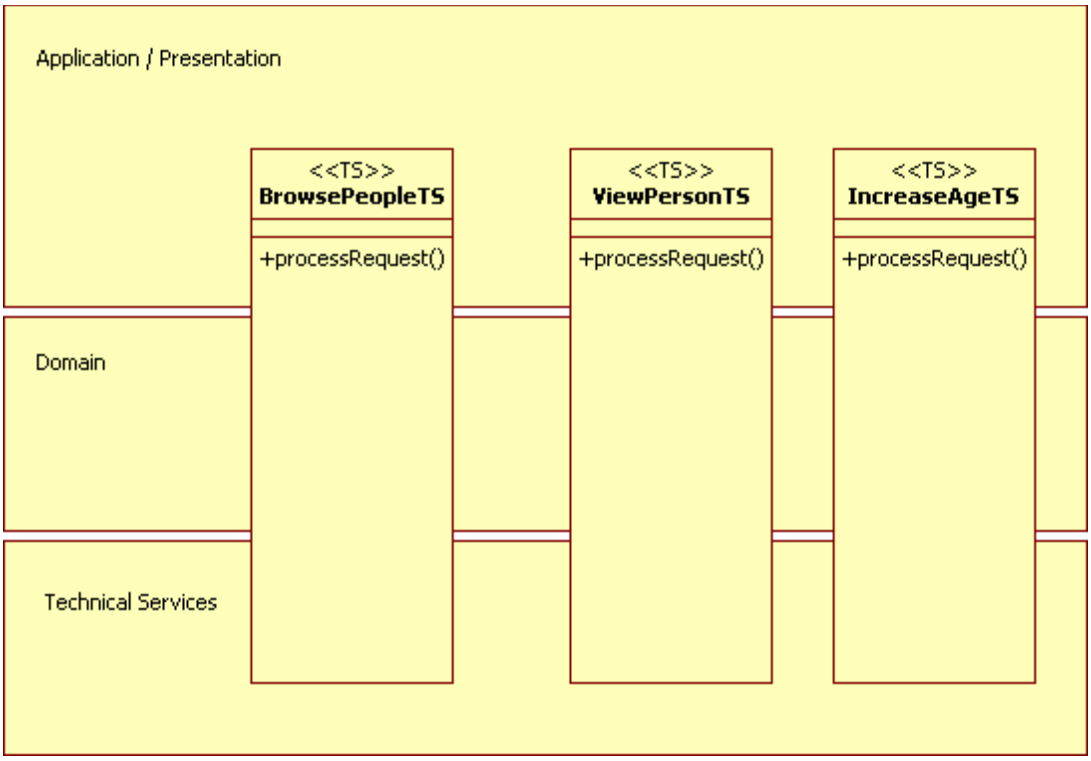


Figure 3-4 Class diagram of TRANSACTION SCRIPTS spanning all three layers

3.2 Iteration Style

We demonstrate an evolution in the use of EA patterns in successive iterations. Each iteration focuses on one or more patterns that increasingly apply separation of concerns or otherwise improve the overall design. The use of iterations puts the patterns in perspective with respect to one another and further provides a basis for our discussion on advanced patterns and pattern usage in later chapters. The iteration-wise breakdown of patterns also allows for a gradual development of familiarity with the patterns.

Each iteration includes a brief summary of the changes from the previous iteration, a class diagram for the current iteration, a description of the relevant patterns, some guidance on the pattern usage and a breakdown of the concerns dealt with by the patterns. In his book on EA patterns, Fowler includes a brief quote at the beginning of each of his pattern chapters. To quickly sum up our exposition of his patterns, and because we find Fowler's quotes to be a good summary, we include these quotes in the "Pattern Description" subsections.

3.3 Iteration 1: Do-It-All TRANSACTION SCRIPTS

In this iteration we provide an implementation to meet the features outlined in Section 3.1: browsing people, viewing people, and increasing their age. This implementation variant of the TRANSACTION SCRIPT pattern is the simplest possible design, and is often called a "Do-It-All" TRANSACTION SCRIPT. Figure 3-4 shows three TRANSACTION SCRIPTS (denoted by the TS stereotype) used by our example application. Each kind of request—one corresponding to each of the features in Section 3.1—has its own class, following a COMMAND pattern approach [Fowler 2003, p.111].

3.3.1 Pattern Description

TRANSACTION SCRIPT

> *"Organizes business logic by procedures where each procedure handles a single request from the presentation."* [Fowler 2003, p.110]

To see how this pattern works, consider what happens after the user issues a request to the server to view all people. The `BrowsePeopleTS` will:

1. Retrieve data about all people from the data source.
2. Format the people data into a suitable response.
3. Return the formatted response to the client.

A TRANSACTION SCRIPT takes responsibility for an entire request.

3.3.2 Pattern Usage

This pattern, especially in the "Do-It-All" form, does little to separate concerns. Unfortunately, it typifies a very commonly used approach to development. This overuse is because a TRANSACTION SCRIPT is easy to implement quickly and requires little thought about program evolution. Throwaway prototypes and experimentation can lead to the use of TRANSACTION SCRIPTs instead of a more structured approach. This pattern captures such ad hoc approaches, the outcomes of which were never designed to be part of a production system but that all too often end up there.

3.3.3 Concerns

TRANSACTION SCRIPTS often deal with the following basic WEA concerns:

- Receive requests from the client.
- Extract client data.
- Apply domain logic.
- Persist results in the data source.
- Generate a response.

In a "Do-It-All" TRANSACTION SCRIPT, all concerns are dealt with in a single TRANSACTION SCRIPT class. TRANSACTION SCRIPTS can also deal with fewer concerns, as we will illustrate next.

3.4 Iteration 2: Isolating Technical Services with ROW DATA GATEWAY (RDG)

Improving on one of the simplest designs possible illustrated in the last iteration, we will introduce a pattern that will help us to separate out the data access concerns from the do-it-all TRANSACTION SCRIPT. In terms of features, the application remains the same.

In Figure 3-5 we see that TRANSACTION SCRIPTs no longer span all three layers, which already makes the design better by more closely adhering to the layered style; as discussed in Section 2.1, classes should be in their own layers instead of spanning several.

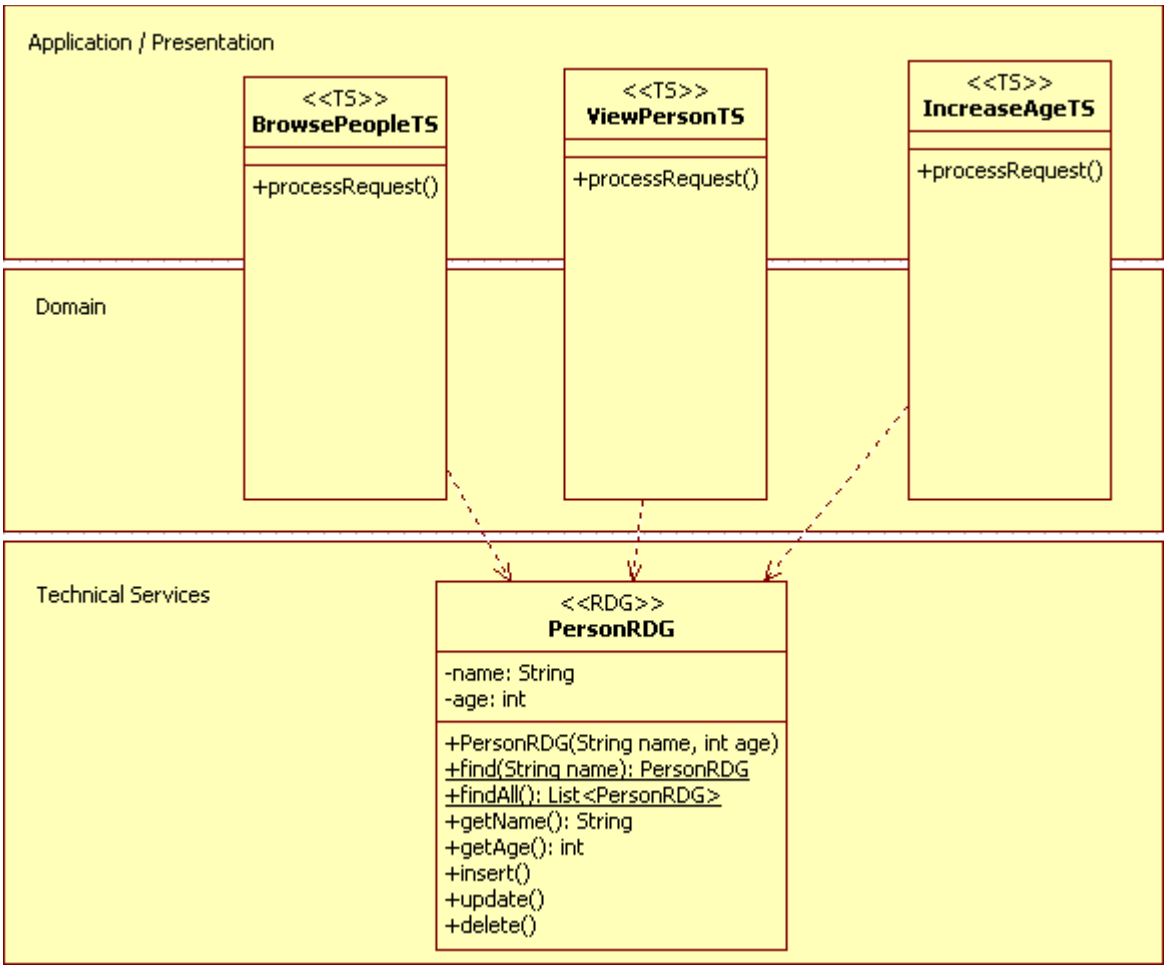


Figure 3-5 Isolating data source access in a ROW DATA GATEWAY

3.4.1 Pattern Description

ROW DATA GATEWAY

> *"An object that acts as a Gateway (466) to a single record in a data source. There is one instance per row."* [Fowler 2003, p.152]

Using RDGs, the data representing a row in the database is stored in an instance of an RDG and is made accessible by standard getters/setters (setters omitted for brevity). As the RDG in our running example has two fields, this implies that the person table in the database has two corresponding columns: name and age.

To interact with the database, RDGs have instance methods insert, update and delete, which affect the corresponding outbound interactions with the data source. To retrieve data from the data source, finder methods are used. These finder methods are often either a part of the RDG or in a separate FINDER class [Fowler 2003, p.152-159, p.161]. RDG FINDERS request data from the data source and instantiate RDGs based on the data found.

3.4.2 Pattern Usage

Fowler recommends the use of RDGs when working with TRANSACTION SCRIPTS or when there is little difference between the representation of data in the application and in the database and when little domain logic is required to be directly associated with the RDGs. Our experience is that RDGs will remain useful longer than TRANSACTION SCRIPTS as a system develops, but eventually become too difficult to work with, as in more complex systems they

either become gradually less cohesive, or become too highly coupled. Even so, RDGs reliably identify an approach used in many systems, and it is a pattern that has merit.

3.4.3 Concerns

ROW DATA GATEWAYs help isolate one concern from TRANSACTION SCRIPTs (Section 3.3):

- Interact with the data source

and adds one new one:

- Stores data from tables.

It is important to note that domain logic is not encapsulated in the RDG. If it were, we would have an ACTIVE RECORD [Fowler 2003, p.155,160], since Active Records are essentially RDGs with domain logic.

3.5 Iteration 3: Isolating Presentation with TEMPLATE VIEW and VIEW HELPER

In the previous iteration we improved the design by factoring out Technical-Services-layer specific concerns into an RDG, leaving the rest of the concerns within the TRANSACTION SCRIPT. In this iteration, we separate presentation concerns from Domain logic while continuing to improve on the design.

In Figure 3-6 we see the addition of two TEMPLATE VIEWs. These additional classes represent a significant separation of concerns, i.e., the removal of the output formatting of the data from the TRANSACTION SCRIPT. The TRANSACTION SCRIPT still contains the Application / Presentation level responsibilities of choosing which VIEW to use (vs. previously generating a response itself) and receiving requests from the client.

TEMPLATE VIEWs are seldom found in a one-to-one relation with the types of requests to an application. In Figure 3-6 we have three types of requests (each TRANSACTION SCRIPT) and only two views. One can easily see why a request to view people is associated with the BrowsePeopleTV and why a request to view a single person is associated with the ViewPersonTV. Figure 3-6 shows that a request to increase the age of a person will be associated with the ViewPersonTV, which allows the application to confirm to the user that they have indeed increased the age of a person.

3.5.1 Pattern Description

TEMPLATE VIEW

> *"Renders information into HTML by embedding markers in an HTML page"* [Fowler 2003, p.350]

The idea behind the TEMPLATE VIEW is to have an otherwise statically generated HTML page embedded with some dynamic content. Java Server Pages (JSPs) provide a mechanism for doing this when working with Java Servlets [JSP, SERVLET].

A goal is to have as little code as possible mixed into TEMPLATE VIEWs, while making them manage as much of the presentation as possible. This usually amounts to mostly static HTML interspersed with calls to getters from a very small number of Domain layer objects, including VIEW HELPERs, which we describe next.

VIEW HELPER

A VIEW HELPER acts, metaphorically, as an envelope to pass data from TRANSACTION SCRIPTs upwards to Application / Presentation layer entities such as TEMPLATE VIEWs. They are often used to aggregate all the pieces used in a VIEW, thus simplifying access. A VIEW HELPER lets us avoid placing domain logic in a VIEW class either by encapsulating this domain logic directly in the VIEW HELPER, or more frequently in our experience, by providing a placeholder for the results of any such domain logic such that the VIEW can just access the result. VIEW HELPERS are also useful to eliminate dependencies from the Application / Presentation layer to the Technical Services layer (thus helping achieve was is called a *pure* form of the layered architectural style [SG96]) and can act as adapters

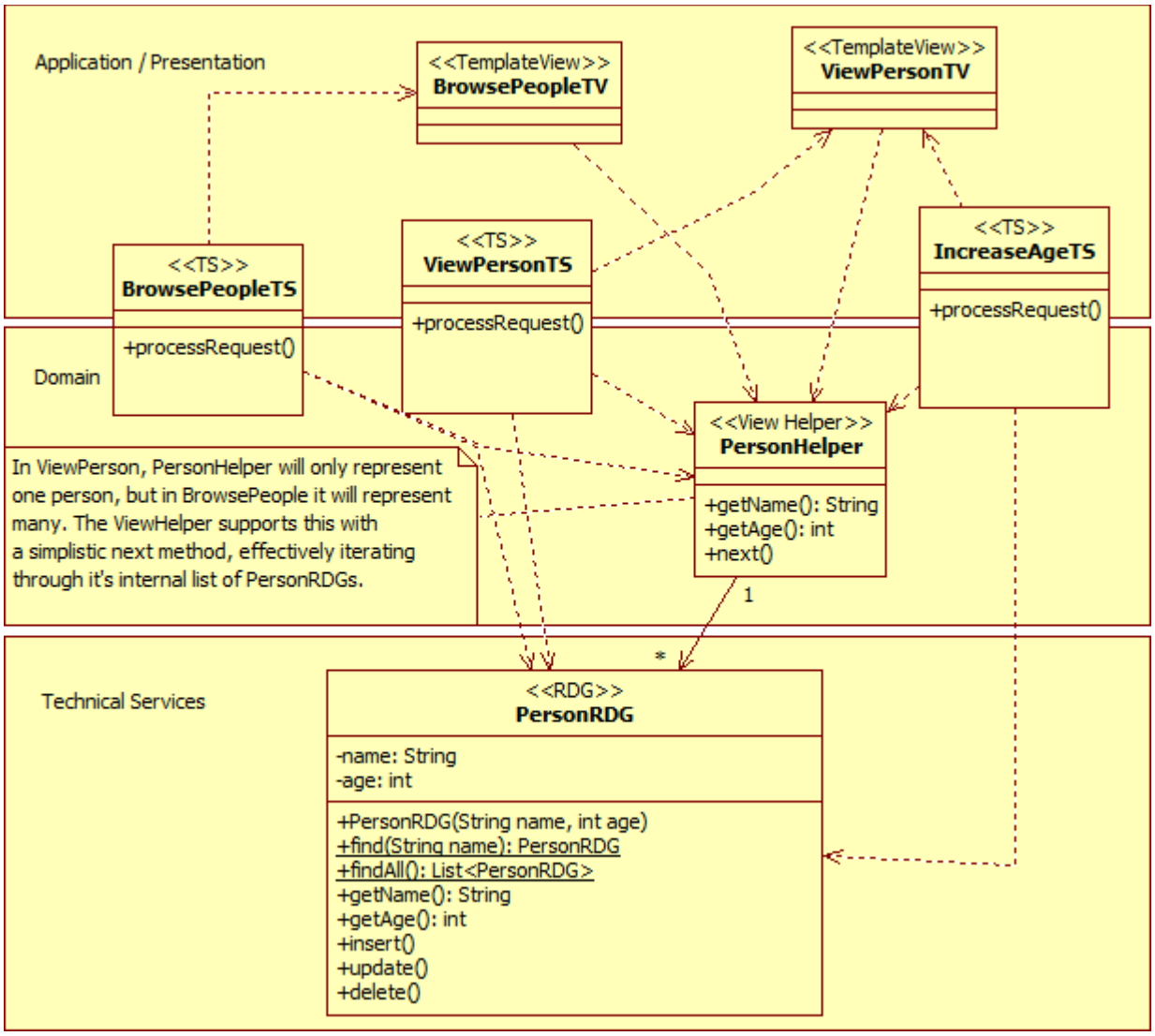


Figure 3-6 Class diagram showing the addition of TEMPLATE VIEWS and a VIEW HELPER

over elements of the Domain layer. For example, by not exposing the RDG class the VIEW HELPER in Figure 3-6 prevents a dependency between the Application / Presentation and Technical Services layers.

Fowler makes use of helpers in his examples, but does not discuss them directly as a pattern in his Patterns of Enterprise Application Architecture (PEAA) book. However, Fowler is a co-author of the Core J2EE Pattern collection [Alur01, p.186] which names this pattern VIEW HELPER—and hence we have also adopted this pattern name. The VIEW HELPER pattern is consistent with Fowler's use of helpers in his PEAA book.

3.5.2 Pattern Usage

TEMPLATE VIEW VS TRANSFORM VIEW

As an alternative to a TEMPLATE VIEW, Fowler also describes the TRANSFORM VIEW. A TRANSFORM VIEW generates content based on the type and format of the data provided, i.e., "transforming" the data from one representation (e.g. XML) to another (e.g. HTML). A TEMPLATE VIEW provides a fixed structure, often with some fixed content, but allowing data to be inserted into predefined locations. Alternately, a TRANSFORM VIEW builds content around data. For the scope of this thesis, the distinction between the two is immaterial. The concerns discussed and the benefits of using either type of view pattern remain the same as far as the rest of the design is concerned.

The TEMPLATE and TRANSFORM VIEWS may not be applicable for all kinds of software applications. For example, if an application's presentation is done directly through a windowed interface (thick-client), then one needs to approach view related responsibilities in an entirely different manner—although the rest of the patterns work quite well in conjunction with thick-client applications.

Although [Fowler 2003] describes both the TEMPLATE VIEW and the TRANSACTION VIEW specifically as patterns to generate HTML, in our experience these patterns are equally well suited to any text-based output.

VIEW HELPER

Being a simple pattern, there is not much to add concerning VIEW HELPER other than they are almost always used when a design includes VIEWS.

3.5.3 Concerns

The TEMPLATE VIEW isolates a single concern from the design of the previous section:

- Generation of output response.

The VIEW HELPER

- Acts as an envelope used to hold data being passed to VIEWS.
- Provides indirection between the Application / Presentation and Technical Services layers.

3.6 Iteration 4: Data Integrity and an Isolated Domain Logic with OPTIMISTIC OFFLINE LOCK, PAGE CONTROLLER and IDENTITY FIELDS

The previous iteration focused on further factoring out concerns from the TRANSACTION SCRIPT. In this iteration we deal with:

- a concurrency problem, and
- a better way of identifying people.

So far, we have considered BuddyAge from the perspective of a single user, but new problems can arise when the application is used at the same time by multiple users. Even the simple case of two people increasing Bob's age at the same time can cause confusing results: namely, one user can see the age of Bob increased by two. To avoid problems like these, we adjust the Increase Age feature so that it notifies users about conflicts due to concurrent updates, as illustrated in Figure 3-7.

In our new design (see Figure 3-8), PAGE CONTROLLERS now occupy the Application / Presentation layer and the remains of TRANSACTION SCRIPTS that used to span that layer and the Domain layer now exist solely in the Domain layer. The design now respects a pure layered architecture. The VIEW HELPER and the RDG now take into account the use of IDs and versions—to be explained shortly. Also, the RDG `update()` and `delete()` methods now return integer values which represent how many records from the data source are affected by those operations.

Time	User 1	User 2
0s	Bob's page is loaded: Bob is 22 years old. Increase This Person's Age Select Another Person	Bob's page is loaded: Bob is 22 years old. Increase This Person's Age Select Another Person
5s	User 1 increases Bob's page: You've increased Bob's age! Bob is 23 years old. Increase This Person's Age Select Another Person	User 2 is still looking at their initial page, they have not done anything yet.
10s	…	User 2 finally decides to Increase Bob's Age: **Warning:** Someone has already modified Bob's age since you last viewed the page. Bob's updated age is given below. Bob is 23 years old. Increase This Person's Age Select Another Person

Figure 3-7 Warning User 2 that someone else has already updated Bob's age

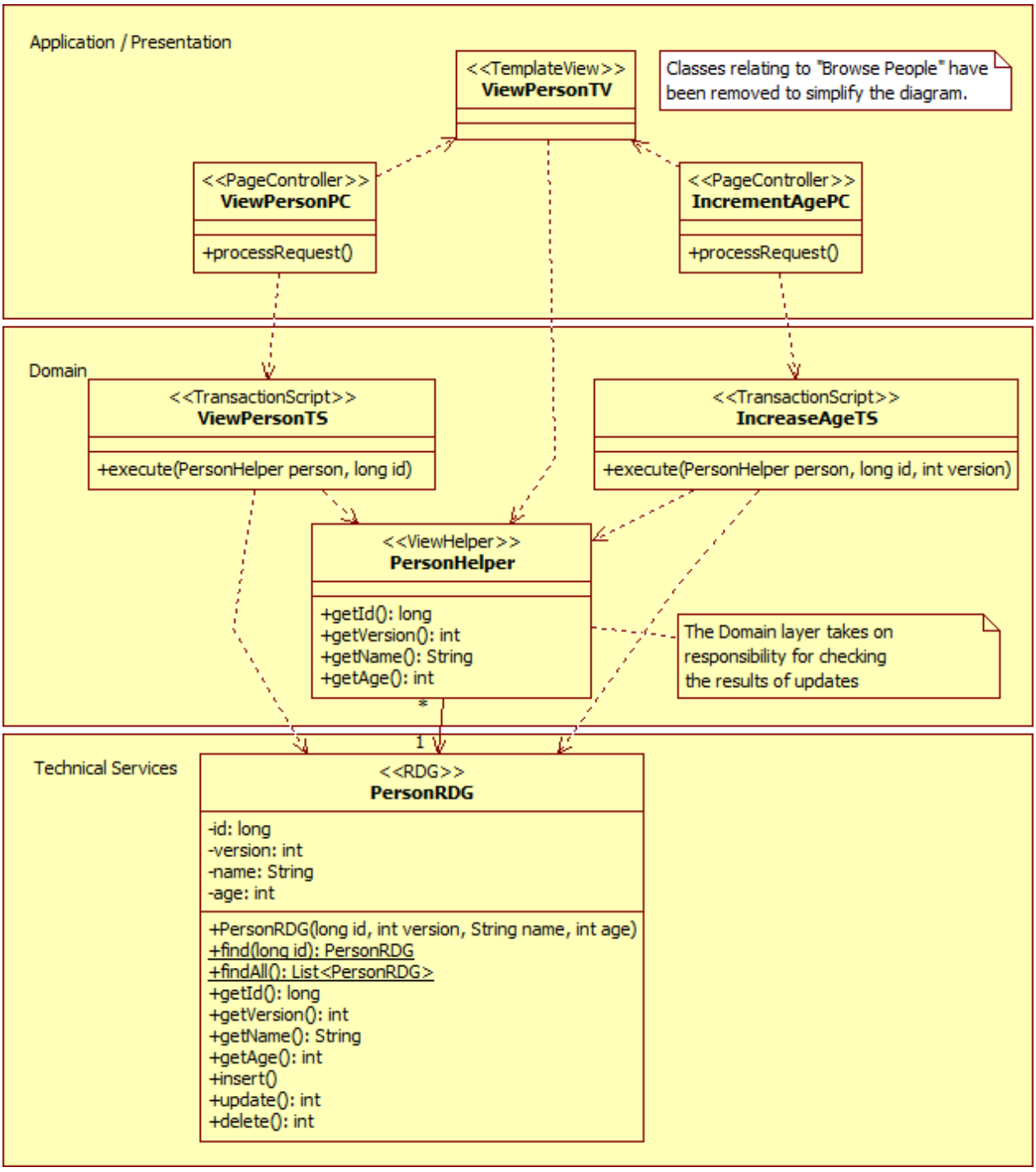


Figure 3-8 A class diagram showing PAGE CONTROLLERS, IDENTITY FIELD and OPTIMISTIC OFFLINE LOCK

3.6.1 Pattern Description

OPTIMISTIC OFFLINE LOCK

> *"Prevents conflicts between concurrent business transactions by detecting a conflict and rolling back the transaction."* [Fowler 2003, p.416]

Concurrency management schemes are characterized by categories at two extremes called optimistic and pessimistic concurrency management. Optimistic schemes do not restrict the ability to change a data source, but help detect conflicts after the fact and focus on the resolution of these conflicts. By contrast, pessimistic schemes eliminate the possibility of conflict by ensuring changes to a data source are serialized; i.e., a pessimistic scheme might allow only one person at a time access to the system, eliminating conflicts with other users. OPTIMISTIC OFFLINE LOCK is best suited to detect lost updates. Lost updates occur when two or more users attempt to change to the same data at the same time, and an earlier change is silently "lost" due to the overwrite of later commits [Fowler05, p.64].

The OPTIMISTIC OFFLINE LOCK pattern works by versioning data in the data source and recording the versions as part of in-memory objects so that when an update is required, the versions of in-memory objects can be compared to those in the data source. If the versions match, the update is allowed. If the versions do not match, the data source is left unchanged and steps are generally taken to notify the user.

Id	Name	Age
1	Bob	22
2	Fred	33
3	Cindy	44

Id	Pet
1	Dog
2	Cat

Thread/Time	**T0**	**T1**	**T2**	**T3**	**T4**
Thread 1	Update Fred (*row locked*)		commit	(*lock released*)	
Thread 2		Update Fred (*blocks, awaiting lock*)			Lost Update Exception

Figure 3-9 Two transactions rendered sequential

Thread/Time	**T0**	**T1**	**T2**	**T3**	**T4**	**T5**	**T6**	**T7**
Thread 1	Starts Transaction	Update Fred (*row locked*)		Update Dog (*blocks, awaiting lock*)				commit
Thread 2	Starts Transaction		Update Dog (*row locked*)		Update Fred (*blocks, awaiting lock*)	**Deadlock detection**	victim rollback (*lock released*)	

Figure 3-10 Deadlock resolution on conflicting transactions

If two update requests affect the same data, the first to go through will acquire a write lock on the data source, most often a row-based lock. Hence, the requests are effectively serialized (see Figure 3-9). Now consider a more complex scenario where two threads each want to update the same pair of rows—Figure 3-10. Conflicts such as deadlock can occur and are detected, reported and one transaction "falls victim" and is rolled back[3].

IDENTITY FIELD

> *"Saves a database ID field in an object to maintain identity between an in-memory object and a database row."* [Fowler 2003, p.216]

In-memory objects do not need any additional means of identifying themselves uniquely until a data source is involved. When using a data source to represent unique in-memory objects, there will be some form of primary key (at worst a compound key comprising all fields in a particular row) which is used as the IDENTITY FIELD. When possible, a simple integer field, not tied to any domain logic, is best. The concept of an IDENTITY FIELD is as simple as it sounds. The only care that must be taken is ensuring a consistent means of getting new IDs. Fowler discusses several methods and the reasoning behind them [Fowler 2003, p.218-220].

[3] Tests confirm that MySQL detects deadlocks of this nature and throws deadlock exceptions.

PAGE CONTROLLER

"An object that handles a request for a specific page or action on a Web site." [Fowler 2003, p.333]

The PAGE CONTROLLER acts as a controller [Larman04, p.288] for the application. With this form of controller, each kind of request has its own entry point into the system in the form of that PAGE CONTROLLER. PAGE CONTROLLERS create and/or initialize Domain layer objects and then apply domain logic as needed (by delegating to Domain objects), finally forwarding results to a VIEW.

3.6.2 Pattern Usage

OPTIMISTIC OFFLINE LOCK

The choice of concurrency management scheme is based on an evaluation of cost. If conflicts are infrequent and the cost of resolving any individual conflict is low, OPTIMISTIC OFFLINE LOCK is a natural choice. In terms of measuring the cost of the conflict, one should consider:

- the cost of automated merging, or
- the cost of offering a convenient UI allowing users to manually merge changes.

IDENTITY FIELD

This pattern always exists in a WEA. In order to be extracted from a data source, there must be some means of uniquely identifying data, though this is not always identified as a key or an IDENTITY FIELD. The actual choice is whether an existing database schema must be used as is, or whether there is flexibility in extending table schemas by adding one or more new columns to serve as primary key. When possible, the use of some form of integer key is best, as integer IDs are simple, are easy to generate sequentially and integer IDs makes for very quick joins of database tables.

```
public class SingleAppUniqueIdFactory extends UniqueIdFactory {

 private static Hashtable<String, Long> IDs = new Hashtable<String, Long>();

 public synchronized long getId(String table, String field) throws SQLException {
  Long max_id = IDs.get(table+"."+field);
  if (max_id == null) {
   ResultSet rs = DbRegistry.getDbConnection().createStatement().executeQuery(
     "SELECT max(" + field + ") AS maximum FROM " +
     DbRegistry.getTablePrefix() + table);
   max_id =  rs.next() ? rs.getLong("maximum") : 1;
   rs.close();
  }
  IDs.put(table+"."+field, ++max_id);
  return max_id;
 }
}
```

Figure 3-11 Single table scan approach

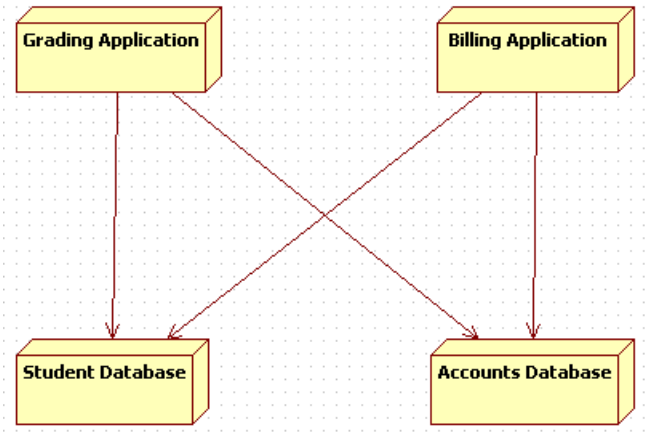


Figure 3-12 Many applications with many databases

Where one gets new ids is dependent on the application. When only a single application is using the database, our approach to generating a unique ID for a new record(s) in a given table, is to perform an initial table scan to obtain the current maximum ID, m, and then using m+1, m+2, ... as the new IDs (Figure 3-11). If multiple applications use the same database, then one must resort to the use of ID tables[4] and a second database connection (which can be used to lock the ID tables). In an environment where there are many databases being shared by many applications (Figure 3-12), a Globally Unique Identifier (GUID) —also sometimes referred to as UUID—system should be used [Fowler 2003, p.218].

PAGE CONTROLLER

Fowler suggests a familiarity of use with PAGE CONTROLLERS as being part of its appeal. Conversely, our approach favors FRONT CONTROLLERS–a pattern described in Section 3.8–to avoid the duplication inherent in using PAGE CONTROLLERS to act as multiple entry points to a single application.

3.6.3 Concerns

No one class represents the OPTIMISTIC OFFLINE LOCK. Managing lost updates, something facilitated by the OPTIMISTIC OFFLINE LOCK pattern, is a concern that impacts other patterns as well, such as RDGs and other data source patterns to be seen in subsequent sections. Detecting concurrency issues is relatively simple while *dealing with them* can be complicated.

In this iteration, the previous TRANSACTION SCRIPT's presentation layer concerns were factored out to PAGE CONTROLLERS. The PAGE CONTROLLER must:

- deal with lost updates, when reported, and

[4] An ID table, in this context, is a table that exists solely to track the current maximums of ids.

- decide on which View to use.

While a PAGE CONTROLLER often only has one choice of VIEW to use, this is not always the case. If there were a separate VIEW dedicated to helping a user resolve a lost update, or a VIEW geared towards displaying on a mobile device, the PAGE CONTROLLER would be responsible for this choice.

3.7 Iteration 5: Refined Access to the Data Source with DATA MAPPER, TABLE DATA GATEWAY and DOMAIN MODEL

In the last iteration we improved upon the means to identify objects and version them. In this iteration we improve upon how we represent and access these objects. The changes in this iteration have no effect on the features of the BuddyAge application.

In Figure 3-13 the PersonHelper and PersonRDG have been replaced with a PersonMapper and Person. The data previously stored in the RDG is now stored in the Person[5], an instance of *part of* what Fowler calls the DOMAIN MODEL pattern. Fowler essentially describes the DOMAIN MODEL pattern as representing the requirements artifact by the same name[6].

The method names for the PersonMapper and the PersonTDG are the same, but while the PersonMapper's methods take a parameter from the Domain Layer, the PersonTDG takes raw data types, representing table fields, as parameters.

[5] Which represents an actual person that has a name and an age.
[6] In [Fowler 2003]'s Domain Model chapter Fowler references a previous edition of [Larman 2004] as his current favorite introduction to OO. Larman's chapter on the Domain Model artifact is very thorough.

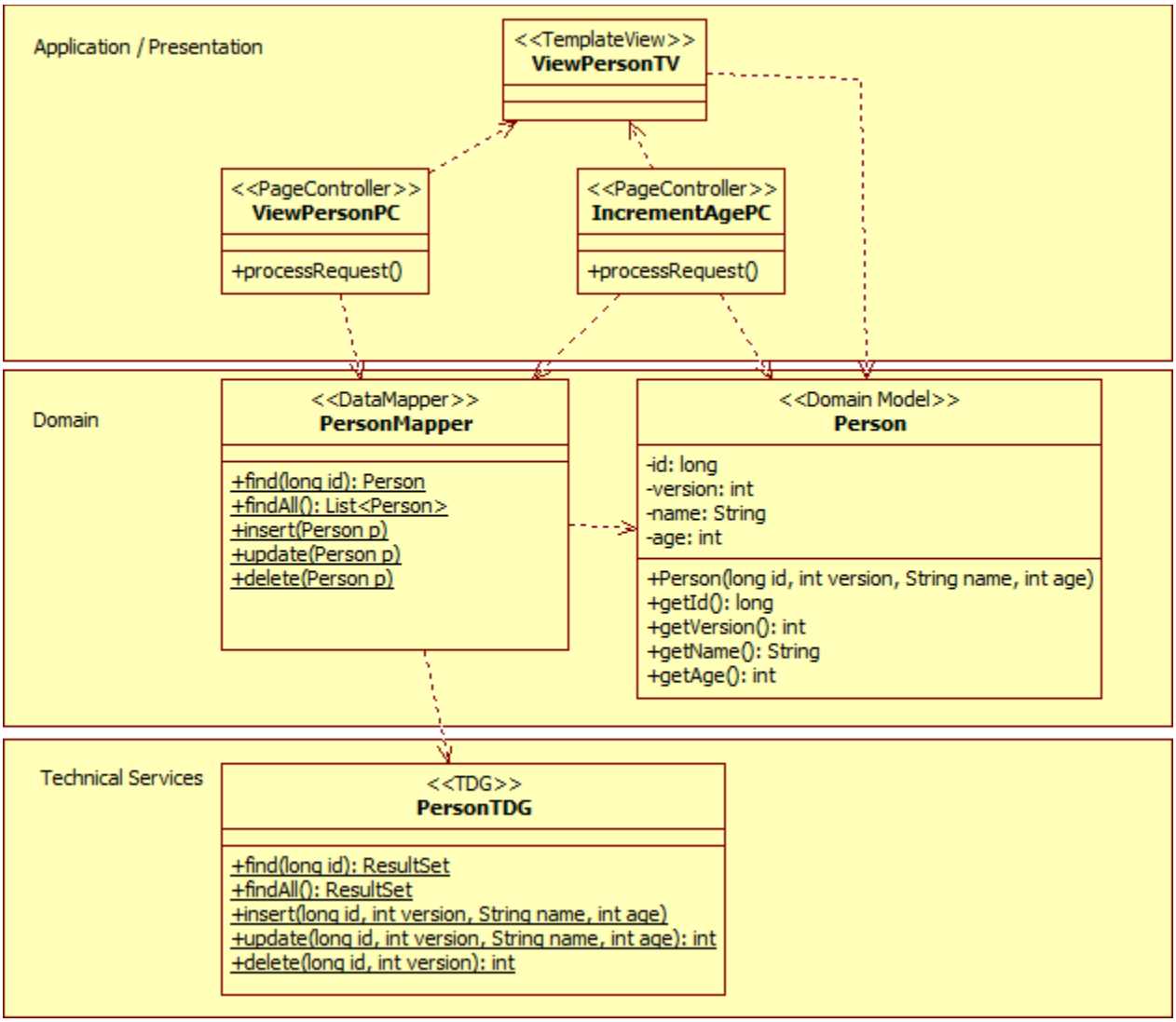


Figure 3-13 Person, PersonMapper and PersonTDG replace PersonHelper and PersonRDG

3.7.1 Pattern Description

DATA MAPPER

> *"A layer of Mappers (473) that moves data between objects and a database while keeping them independent of each other and the mapper itself."* [Fowler 2003, p.165]

A DATA MAPPER allows the differences between the domain and the data source to be isolated. In-memory objects may map to many columns, involve inheritance or have complex relationships with other in-memory objects. The many approaches used to deal with the differences between the data source and in-memory representations of Domain data are designed into DATA MAPPERS. These include deciding how much data to load from the database, keeping track of loaded data to prevent duplicate loading, eliminating cyclic dependencies of loaded data, loading data from multiple sources and persisting data from one in-memory object to multiple tables in the data source.

In-memory representations of data obtained from the data source are no longer associated with the fact that they come from a data source. This applies a fundamental Object-Oriented design principle, the separation of concerns, by encapsulating how fields of the in-memory representations are converted to/from the data source in the DATA

MAPPER. The immediate gain is the ability to intuitively represent much richer data, including associations between in-memory representations of data from the data source—a significant improvement over what could be done with RDGS.

TABLE DATA GATEWAY

> *"An object that acts as a Gateway (466) to a database table. One instance handles all the rows in the table."* [Fowler 2003, p.144]

The TDG is simply a means to isolate SQL from the rest of the code. TDGs have static methods for standard CRUD database access: several methods to retrieve data, a method to create rows of data, a method to update rows of data and a method to delete rows. In cases where the TDG represents a view on a table, there may only be methods to retrieve data.

DOMAIN MODEL

> *"An object model of the domain that incorporates both behavior and data.."* [Fowler 2003, p.116]

[MartinFowler] provides a succinct description: "A Domain Model creates a web of interconnected objects, where each object represents some meaningful individual, whether as large as a corporation or as small as a single line on an order form."

3.7.2 Pattern Usage

DATA MAPPER

Fowler suggests the use of the DATA MAPPER pattern whenever the data source's organization and in-memory organization evolve independently [Fowler 2003, p.170]. He also suggests that the DATA MAPPER can be avoided if "the domain model is pretty simple, and the database is under the domain model developer's control". We find that the organization of data for a system of any complexity quickly evolves to the point where a DATA MAPPER is needed, so we always advocate its use.

TABLE DATA GATEWAY

Fowler suggests that the use of DATA MAPPER subsumes the role of TDG, leaving the DATA MAPPER spanning two layers. Larman, in contrast, considers that TDGS can be complementary to DATA MAPPERS, placing each in a separate layer, and suggests using TDGS to isolate the SQL—which is the approach we use here.

DOMAIN MODEL

Fowler suggests that DOMAIN MODEL is used when an application becomes complex enough, We discuss our approach to the DOMAIN MODEL pattern in Section 4.1.

3.7.3 Concerns

The DOMAIN MODEL pattern helps lower representational gap between conceptual representation and corresponding implementation.

Helpers like the PersonHelper in Figure 3-8 are replaced by classes that contain data directly, instead of using underlying data access elements like RDGS. In doing so, one must separate the storage of in-memory data from interaction with a data source, hence new concerns are highlighted for the DATA MAPPER:

- Mapping data from the data source to in-memory objects;
- Persisting, to the data source, new in-memory objects and changing existing ones.

TDGs have one obvious concern, the isolation of SQL. While not mentioned explicitly in Fowler, his examples show careful attention to sanitizing outbound data, which is a TDG's second concern.

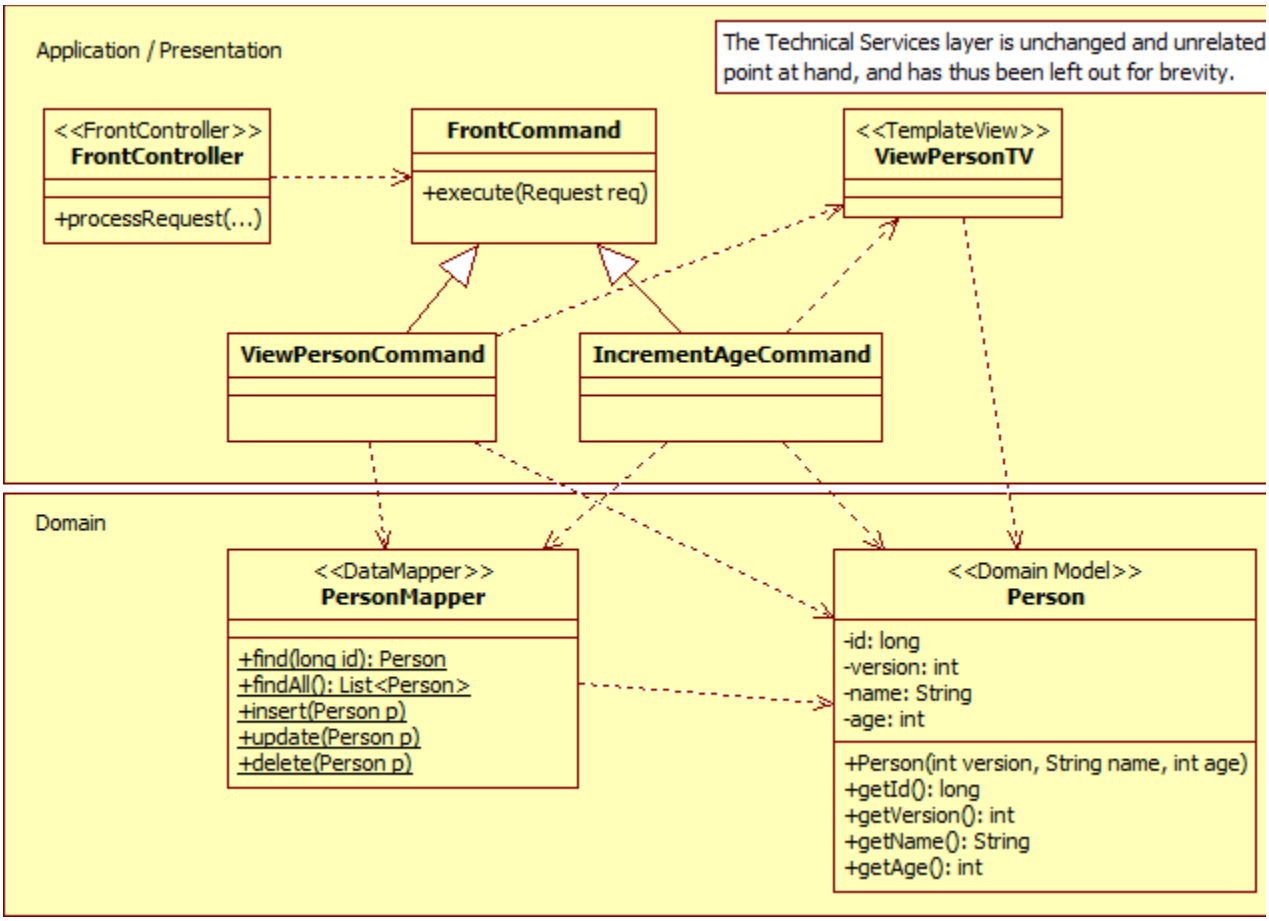


Figure 3-14 Using a FRONT CONTROLLER and COMMANDS to replace PAGE CONTROLLERS

3.8 Iteration 6: An Organized Approach to the Application Layer Using the FRONT CONTROLLER Pattern

The last iteration improved how we represented and accessed data, dealing with the Domain and Technical Services layers. This iteration will factor out the Application components of the system, continuing to keep the same overall features for the BuddyAge application.

In Figure 3-14, the PAGE CONTROLLERS are replaced by FRONT COMMANDS, a variant on the COMMAND pattern [GoF]. An abstract `FrontCommand` allows the newly added FRONT CONTROLLER to have no direct dependency on the concrete FRONT COMMANDS. The FRONT CONTROLLER replaces most of the code that was duplicated across the original PAGE CONTROLLER classes, specifically the initialization and the incoming request mechanism.

3.8.1 Pattern Description

FRONT CONTROLLER

> *"A controller that handles all requests for a Web site."* [Fowler 2003, p.344]

FRONT CONTROLLERS handle all incoming requests to a WEA and dispatch the requests to an appropriate FRONT COMMAND. FRONT CONTROLLERS can instantiate FRONT COMMANDS through various means: a simple if/else tree, looking up the FRONT COMMAND in a database or properties file based on a provided key, or sending that key to a factory which does the lookup. In environments that support reflection, if the key is a fully qualified class name, the

appropriate FRONT COMMAND can be directly instantiated using the reflection mechanism [Fowler 2003, p.348]. Fowler describes a FRONT COMMAND as the part of the FRONT CONTROLLER pattern that redirects to the desired VIEW after its processing of domain logic is complete.

3.8.2 Pattern Usage

FRONT CONTROLLER

The FRONT CONTROLLER pattern is used by a system as a single point of access, which makes sense if the goal is to isolate most of the environment-specific elements of a WEA. This approach is commonly used in a wide variety of open source PHP/CGI applications, although it is not explicitly named. Users access a single page with different parameters and the web application does different things accordingly.

FRONT CONTROLLERs can also be mixed with PAGE CONTROLLERs or split into mini-FRONT CONTROLLERs, each covering a different part of the application's functionality, or a separate subsystem [Larman 2004]. One could imagine one FRONT CONTROLLER taking care of all the publicly available functionality for a system, thus foregoing security checks, while a sibling FRONT CONTROLLER takes care of the secured behaviors. Reasons for doing so are dependent on topics well outside the scope of this thesis, however we promote the use of a single FRONT CONTROLLER.

3.8.3 Concerns

Refactoring of PAGE CONTROLLERs into a FRONT CONTROLLER and FRONT COMMANDS simplifies the application design and also renders it more adaptive: i.e., the CONTROLLER no longer needs to know explicitly what sorts of requests it is capable of serving, as this knowledge can be inferred at runtime. The FRONT CONTROLLER is instead tasked to:

- receive requests from the client,
- redirect to FRONT COMMANDS supplied in the request.

The PAGE CONTROLLERS are otherwise turned into FRONT COMMANDS which

- Extract client data,
- Apply any domain logic,
- Decide on which VIEW to use.

A FRONT CONTROLLER may also extract data from requests, but only so much as to allow the appropriate FRONT COMMAND to be determined.

In more advanced applications, the FRONT CONTROLLER, being the single entry point, often appears to take on other concerns. One finds various application level configurations for logging and database configuration as well as some per-request preparation for things like database transactions, file upload preparation or the cleaning out of ThreadLocals[7] in a shared thread environment (like in Tomcat 5 or later releases). These concerns are not part of the FRONT CONTROLLER pattern, but they often appear in its implementation.

3.9 Iteration 7: Managing In-Memory Data with LAZY LOAD (via VIRTUAL PROXY) and IDENTITY MAP

The previous two iterations focused on organizing basic web application concerns cleanly across the three-layered scheme. In this iteration, the focus will be on patterns that provide guidance for dealing with some common WEA issues such as loading the same data into memory more than once or loading data that contains cyclic references.

[7] Declaring a field as ThreadLocal makes it appear to be unique to each thread. Behind the scenes, a ThreadLocal field access behaves like accessing a field that is a hash table keyed on the id of the current thread.

To highlight the issues, we will change the system slightly. A `Person` can now have a buddy who may be any other `Person` in the system. So, as far as a user of BuddyAge is concerned, when viewing a `Person`, that `Person`'s buddy will be shown (Figure 3-15).

In Figure 3-16, we see the trivial case of how a cyclic reference can happen. While it appears easily avoidable, one can imagine a more complex and less direct cyclic reference involving more than two objects. In order to be able to create an Alice object we need a reference to Bob, but we cannot create Bob unless we first have a reference to Alice. While code could be written specifically to handle this situation, LAZY LOAD provides a simpler and uniformly applicable solution regardless of the complexity of the object interdependencies.

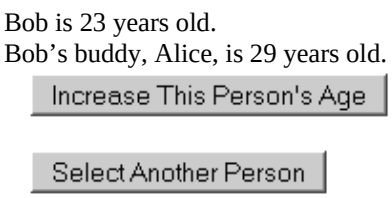


Figure 3-15 Viewing a Person and their buddy

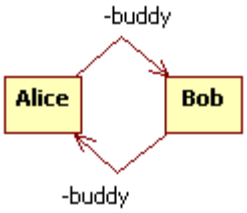


Figure 3-16 Alice is Bob's buddy. Bob is Alice's buddy

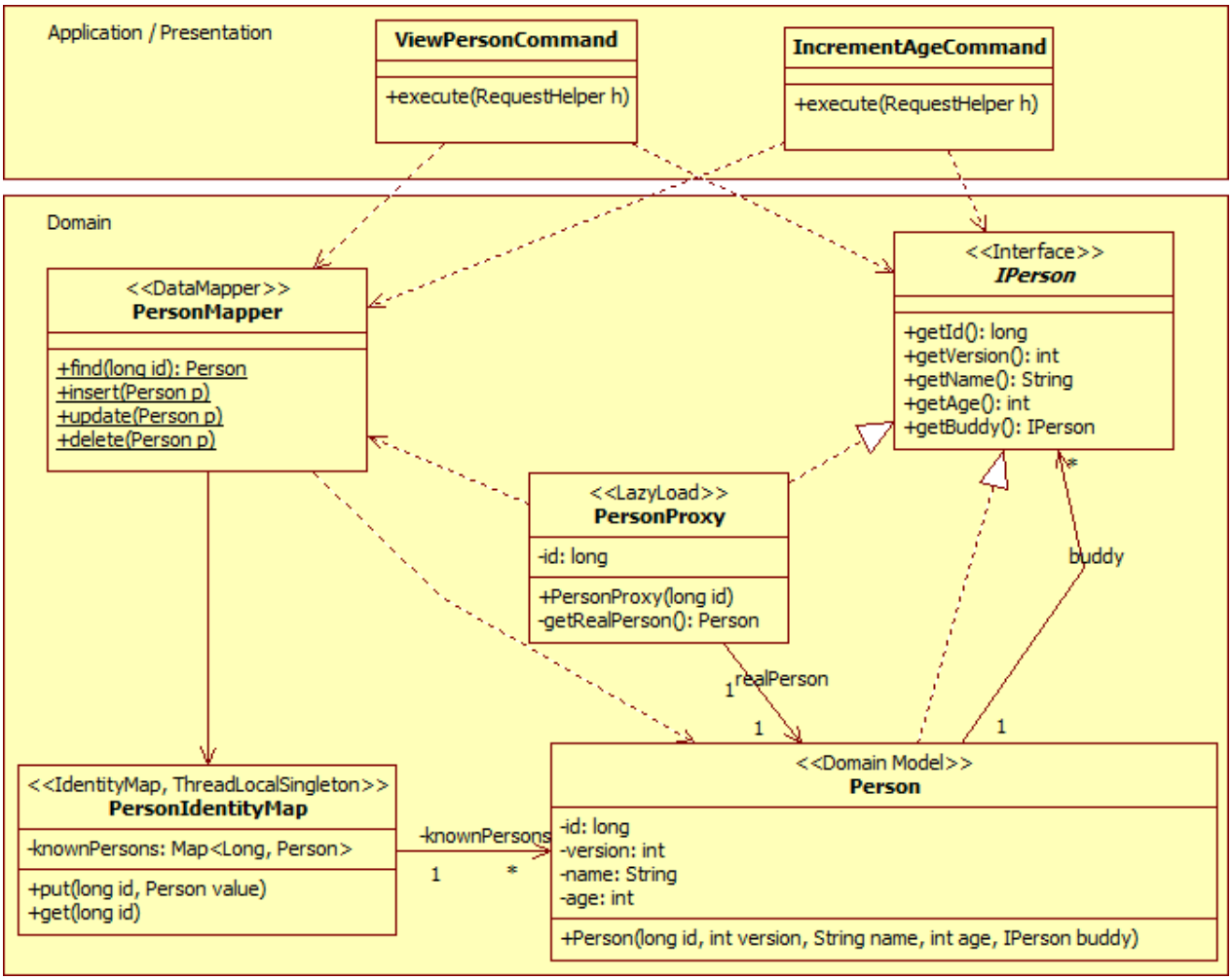


Figure 3-17 Adding a VIRTUAL PROXY and IDENTITY MAP

LAZY LOAD via VIRTUAL PROXY is illustrated in Figure 3-17: the interface of Person has been factored out into the IPerson and a PersonProxy has been added. IPerson abstracts away whether a Person or a PersonProxy is being used. A PersonProxy can stand in for a Person without actually needing any of that Person's data, except the IDENTITY FIELD. Figure 3-18 shows how the VIRTUAL PROXY finds an actual Person using the person's `id`; the rest is done with delegation.

The next change to the class diagram is the addition of an IDENTITY MAP. Whenever the PersonMapper is tasked with finding a Person, it will first query the IDENTITY MAP to see if the Person has already been loaded, and if not, it loads the Person and stores it in the IDENTITY MAP. IDENTITY MAP's are unique to each request, which is guaranteed by the ThreadLocalSingleton stereotype.

3.9.1 Pattern Description

LAZY LOAD

> *"An object that doesn't contain all of the data you need but knows how to get it."* [Fowler 2003, p.200]

Fowler describes the problem of loading an object from a heavily interconnected database. If each object loaded into memory loads up all associated objects, one could conceivably end up with an entire database loaded. The key idea is that rather than have an object getting fully loaded into memory, it will instead get loaded into memory only when it is needed. Larman describes loading the entire object into memory immediately as "eager" loading, also using the term "lazy" to describe when the object is not fully loaded until used [Larman 2005].

Fowler does not explicitly discuss cyclic references in his description of Lazy Load. We have found that it happens more readily than the loading of too much data, so we suggest that this is the primary problem addressed by Lazy Loading, and will discuss the matter further in Section 4.3.

IDENTITY MAP

> *"Ensures that each object gets loaded only once by keeping every loaded object in a map. Looks up objects using the map when referring to them."* [Fowler 2003, p.195]

Problems can arise if an object is loaded into memory more than once and one instance is changed in one way, while another instance is changed in a different way. Correctness can not be guaranteed if both those changes were written to the database.

3.9.2 Pattern Usage

LAZY LOAD

Fowler suggests three variants of the LAZY LOAD pattern. Of his suggested variations, the VIRTUAL PROXY is the cleanest, in our opinion. Another common approach is to use a GHOST, wherein an object can be partially loaded. This partially loaded object would then load itself the rest of the way as needed. A GHOST essentially acts as a VIRTUAL PROXY for itself. There is also a variation on LAZY LOAD called LAZY INITIALIZATION [Larman 2004] where `null` is assigned to a field. Any access to that field checks if it is null, loading the real data if the field was `null`.

```
private Person getRealPerson() {
  if(realPerson == null) {
    realPerson = PersonMapper.find(id);
  }
  return realPerson;
}
```

```
public int getAge() {
  return getRealPerson().getAge();
}
```

Figure 3-18 PersonProxy code for getting a real Person and illustrating delegation

Fowler advocates that LAZY LOAD should be used as dictated by performance, and sparingly at that. We strongly advocate the use of LAZY LOAD even when performance does not explicitly indicate that it would be needed. While our reasoning will be explained fully in Chapter 4, at this stage the avoidance of problems arising from cyclic dependencies alone is a very strong reason to use this pattern.

As mentioned earlier, we advocate the use of VIRTUAL PROXY as the preferred variant of LAZY LOAD. Both GHOST and LAZY INITIALIZATION create incomplete objects that must be aware of a means to load additional data. With VIRTUAL PROXY, objects remain oblivious to the fact that their data is not completely loaded, as well as to how that data would be retrieved, thus making for less coupling and higher cohesion.

IDENTITY MAP

The IDENTITY MAP is required in any system where data is retrieved from a data source, changed, and saved again. Specifically, anything in memory making use of the IDENTITY FIELD pattern should be used with an IDENTITY MAP. Fowler also suggests that IDENTITY MAPs act as a reading cache, saving external calls to a data source. Our experience has shown that this use is beneficial in a wide variety of systems, even though it is a side effect of the main benefit of the IDENTITY MAP.

3.9.3 Concerns

The concerns dealt with by these two patterns are of a different nature than those we have discussed thus far. The LAZY LOAD pattern:

- ensures that problems arising from cyclic dependencies can be avoided,
- prevents a system from using resources it does not need.

The IDENTITY MAP pattern:

- ensures that an object only gets loaded from the data source into memory at most once per request,
- potentially helps reduce the number of calls to the persistent store.

3.10 Iteration 8: Accommodating a Complex Domain with UNIT OF WORK (UOW) and DEPENDENT MAPPING

In this iteration we change how the BuddyAge application works. Now each Person will have several PhoneNumbers associated with him/her. These PhoneNumbers are only relevant insofar as they are related to a Person. In Figure 3-19 we see that the additional PhoneNumber class has a many-to-one relationship with Person.

The code for the PersonMapper's find, insert, update and delete methods is given in Figure 3-20. It illustrates the new responsibilities assigned to PersonMapper with respect to keeping track of a person's PhoneNumbers. Since PhoneNumbers do not have an independent identity, we use the DEPENDENT MAPPING pattern, embodied in the PersonMapper methods, where:

- inserting a person causes their phone numbers to be inserted (line 30),
- deleting a person causes their phone numbers to be deleted (line 39),
- updating a person causes their phone numbers to be deleted and reinserted (line 34-35) and
- loading a Person into memory causes their PhoneNumbers to also be loaded (line 16-20).

There is no need for an IDENTITY MAP when using DEPENDENT MAPPING on PhoneNumbers as they lack identity.

Figure 3-20 also shows a UNIT OF WORK with the principle function of keeping track of changes to in-memory objects. As a Person is changed, removed or added, this change will be registered with the UNIT OF WORK (line 23). When a Command is finished processing, it will use the UNIT OF WORK to commit all the registered objects.

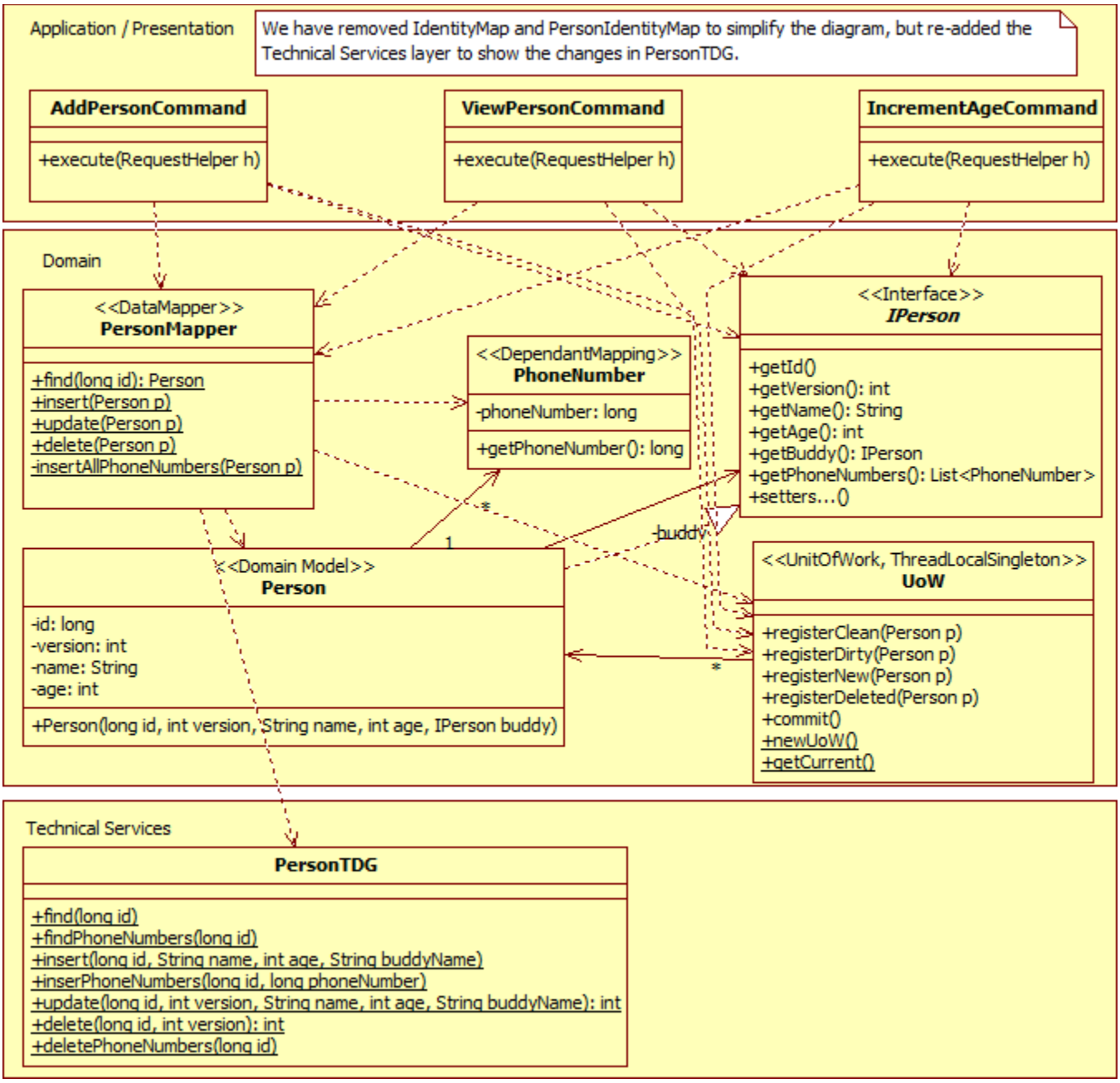


Figure 3-19 UoW and a relationship between Person and PhoneNumber

```
1  public static Person find(long id) {
2    if(PersonIdentityMap.has(id))
3     return PersonIdentityMap.get(id);
4    Resultset rs = PersonTDG.find(id);
5    if(rs.next()) {
6     List<PhoneNumber> numbers =
7      new Vector<PhoneNumber>();
8     Person p = new Person(id,
9      rs.getLong("p.version"),
10     rs.getString("p.name"),
11     rs.getInt("p.age"),
12     new PersonProxy(rs.getLong("p.buddy")),
13     numbers
14    );
15    rs.close();
16    rs = PersonTDG.findPhoneNumbers(id);
17    while(rs.next()) {
18     numbers.add(
19      new PhoneNumber(rs.getLong("pn.number")));
20    }
21    rs.close();
22    PersonIdentityMap.put(id, p);
23    UoW.registerClean(p);
24    return p;
25   }
26   return null;
27 }
```

```
28 public static void insert(Person p) {
29   personTDG.insert(...);
30   insertAllPhoneNumbers(p);
31 }
32 public static void update(Person p) {
33   personTDG.update(...);
34   personTDG.deletePhoneNumbers(p.getId());
35   insertAllPhoneNumbers(p);
36 }
37 public static void delete(Person p) {
38   personTDG.delete(...);
39   personTDG.deletePhoneNumbers(p.getId());
40 }
41 private static void
42  insertAllPhoneNumbers(Person p) {
43   for(PhoneNumber phoneNumber: p.getPhoneNumbers()) {
44    personTDG.insertPhoneNumber(p.getId(),
45     phoneNumber.getPhoneNumber());
46  }
47 }
```

Figure 3-20 PersonMapper methods

3.10.1 Pattern Description

UNIT OF WORK

> *"Maintains a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems."* [Fowler 2003, p.184]
>
> *If data is changed in-memory, it needs to be persisted to the data source for other requests to see it. If not done systematically, keeping track of what is changed is difficult as a system becomes larger. Alternatively, writing out changes frequently can be slow and impractical; there may be many changes, and an actual transaction opened in a data source may need to persist across multiple requests [Fowler 2003, p.184].*

The UNIT OF WORK addresses these problems by tracking the state of four types of in-memory data:

- Clean: the object is in the data source and the in-memory data is consistent with the data source data;
- New: the object is not yet in the data source;
- Removed: the object should be removed from the data source;
- Dirty: the object has changed from what was retrieved from the data source.

The UNIT OF WORK supports a business transaction by tracking such categorized data: recording changes (and possibly reads), starting the transaction, performing concurrency checks and then writing the changes to the data source all at once [Fowler, p.185].

Fowler describes the Unit of Work as also being the place to resolve the two related technical problems:

- Maintaining referential integrity (relative to foreign keys)
- Avoiding deadlocks

A data source has referential integrity if all foreign keys refer to entries that exist. E.g., imagine a `Person` table and a `Pet` table. If `Pet` has a foreign key on `Person`, perhaps owner, then a `Pet` must always have a corresponding row in the `Person` table. If one needs to delete a row from `Person`, and the corresponding row(s) from `Pet`, one must

be careful of the order of database requests to ensure referential integrity, i.e. if the `Person` row is deleted first, referential integrity is violated.

A form of lost update, for row-based locking, can cause deadlocks as covered in Section 3.6.1, particularly Figure 3-10. Table-based locking would simply make this a bigger problem. Fowler suggests that logic organizing the order of actual data source interactions is a means to minimize both deadlock and referential integrity issues, and that such logic belongs in a UNIT OF WORK. Explicit ordering mechanisms are not given as part of UNIT OF WORK, but a suggestion of topological sorting is made [Fowler, p.188].

DEPENDENT MAPPING

> *"Has one class perform the database mapping for a child class."* [Fowler 2003, p.262]

Some data does not need, and often does not have, unique identifiers. The DEPENDENT MAPPING pattern deals with this situation. DEPENDENT MAPPINGs do not need versions or IDs, they depend on the version and ID of the data that references them.

3.10.2 Pattern Usage

UNIT OF WORK

The UNIT OF WORK pattern is simple to use and can provide tangible performance benefits with very minimal implementation or design effort. If a UNIT OF WORK implementation accepts different types of object, it makes it even easier to continue to use it as a system evolves. Fowler also points out that it effectively scales to support concurrency management [Fowler 2003, p.190], and while there are other means of keeping track of changes to in-memory data, UNIT OF WORK is arguably the simplest to use.

UNIT OF WORK is always applied within the context of a single request. Fowler discusses the possible use of UNIT OF WORK across multiple requests, without offering details. We consider spanning multiple requests with a UNIT OF WORK to be non-trivial and outside the scope of this thesis. Furthermore, Fowler suggests that IDENTITY MAP can be bundled into UNIT OF WORK by recording reads as well as other changes, an approach that we endorse.

DEPENDENT MAPPING

A DEPENDENT MAPPING should be used whenever a Domain Model identifies dependent data that is exclusively referenced by some primary data. For example, if a system keeps track of a person's phone numbers (e.g. mobile, home, work, etc.), this could be a candidate for a DEPENDENT MAPPING.

Conversely, if a person and their significant other are in a given system, and they share the same pool of phone numbers, phone numbers always changing for both of them at the same time, then the dependent data would no longer be exclusively referenced by one person and such a system would *not* be a candidate for DEPENDENT MAPPING.

Dependent data does not have identity beyond that of its primary data, but it can always be looked up using its primary data.

3.10.3 Concerns

The UNIT OF WORK and DEPENDENT MAPPING patterns mostly help contribute to simplified implementations. The UNIT OF WORK prescribes a way of thinking about in-memory objects that is in-line with the CRUD approach to dealing with data. Consequently, UNIT OF WORK:

- groups all changes to the database for a request, and also
- provides a means to manage concurrency

The DEPENDENT MAPPING pattern does not so much separate a concern as it identifies an organization of data in the database, so we will not assign a specific concern to that pattern.

4 WEA Design Patterns Revisited

In the previous chapter we looked at some of the key patterns described in [Fowler 2003] and how they can be used in conjunction with each other. This chapter presents a refinement of the patterns described in the previous chapter, as well as some additional design patterns that have become apparent given our experiences teaching WEA design patterns and applying them to sizeable academic and commercial applications.

4.1 DOMAIN OBJECT

This section presents a pattern which identifies how real world concepts can be translated into programmatic equivalents that address common identity and concurrency issues.

4.1.1 Context

DOMAIN OBJECT and Business Object[8], often used interchangeably, are terms that we have frequently come across in EAs. These kinds of objects are instances of DOMAIN MODEL classes. "Domain Object" (as the name of a class) is also used in the context of EAs, most often to mean a class that represents a LAYER SUPERTYPE[9] to Domain Model classes. To our knowledge however, no other author has formally defined DOMAIN OBJECT as a pattern. We provide such a definition for DOMAIN OBJECT in this section.

4.1.2 Problem

Maintaining identity and consistency in a DOMAIN OBJECT that is represented both in memory and in persistent storage can be difficult. In particular, a DOMAIN OBJECT represents data that needs to be isolated and to have identity because it should be manipulated by the system as a single unit. Furthermore, concurrent access of DOMAIN OBJECTS is extremely common, which leads to the general problem of Lost Updates and Inconsistent Reads [Fowler 2003, p.64].

4.1.3 Solution

The first step is to isolate the data along conceptual boundaries. This can be done be identifying the elements representing abstract classes in a Domain Model diagram. Describing these elements in terms of DOMAIN OBJECTS provides an identity that remains with the associated data both in memory and in the persistent storage, which allows for clearer identification of when either of these two representations have changed.

For example, one can have a particular `Person`, uniquely identified by the ID 1 and having a version 1. This `Person` can be stored, changed, retrieved and otherwise accessed concurrently. This `Person`, instantiated, could represent Bob, who is 18 years old. A program can change Bob to be 19 years old, but this would necessarily change the version to 2. Another program, still thinking Bob is 18 (having version 1 in memory) would be able to identify that its version of Bob is not as current as the version of Bob that is 19 (a similar example was covered in Figure 3-7). This means that it is possible for two programs using the same WEA to have different versions of the same data.

The primary advantage is that concurrency management can be encapsulated within the DOMAIN OBJECT. A secondary advantage is that DOMAIN OBJECTS now have an explicit meaning for non-developers that is consistent with its meaning for developers.

[8] "Technically, business objects encapsulate traditional lower-level objects that implement a business process (i.e., they are a collection of lower-level objects that behave as single, reusable units). User interfaces can be thought of as views of large-grained Business Objects. Databases maintain a record of the "state" of Business Objects as they change over time." [Sutherland97]

9 Layer Supertype is a pattern from [Fowler 2003]

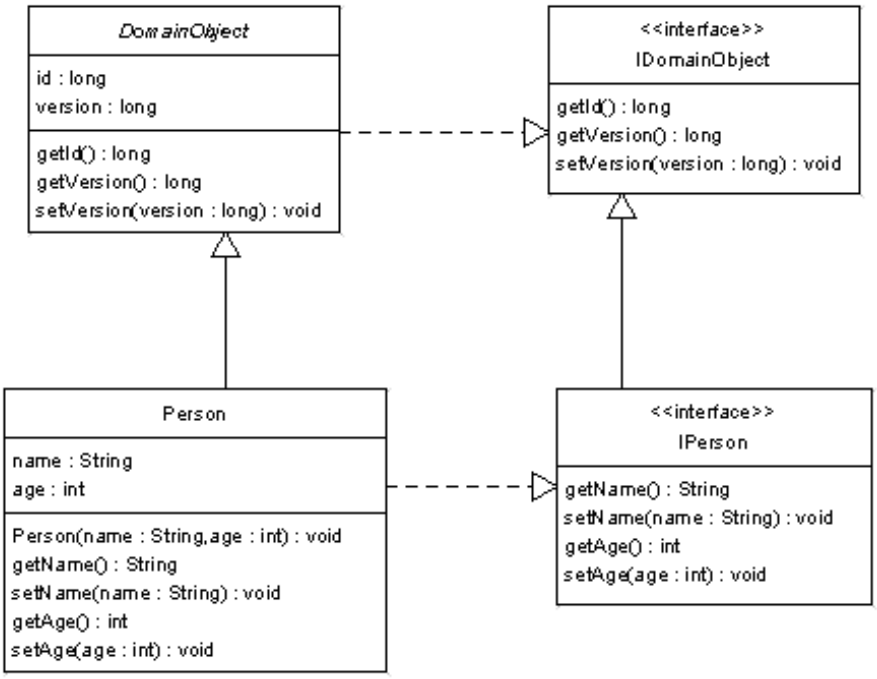


Figure 4-1 An implementation of Person using the Domain Object Pattern

Figure 4-1 illustrates the use of interfaces during the implementation of a system that uses the DOMAIN OBJECT pattern. Using interfaces in this way is a well established programming practice to separate behavior from implementation. In this case, the implementation of the Person is made clear: it uses a LAYER SUPERTYPE [Fowler 2003 p.475] which contains ID and version data. Instances of such a concrete implementation are often considered the "real" in-memory DOMAIN OBJECTs. This practice should always be considered when implementing a system that applies the DOMAIN OBJECT pattern.

Developers and other stakeholders do not need to know if a DOMAIN OBJECT instance is of a particular type of that DOMAIN OBJECT, or if it is a PROXY (LAZY LOAD [Fowler 2003 p.200] pattern via VIRTUAL PROXY), as an interface such as `IPerson` would hide this information. The particulars of implementation out of the way, stakeholders have new terminology that is valuable in its familiarity, and made effective by its low representational gap. For example, when a developer talks about a programmatic `Person` and an end user talks about a real Person using the system, they can overlook the fact that they are talking about different concepts[10].

4.1.4 Related work and contribution

Related work:

- Fowler [Fowler 2003] used the term DOMAIN OBJECT without an explicit definition.
- DOMAIN OBJECTS have been loosely associated with "Business Objects"[11].
- Fowler [Fowler 2003] used a `DomainObject` LAYER SUPERTYPE in some examples.
- DOMAIN OBJECTS have been described as part of the DOMAIN MODEL discussed in Section 3.7.

Our contribution:

- Identify DOMAIN OBJECT as a pattern.
- Require DOMAIN OBJECTS to have unique IDs

[10] Thomas Triplet (http://www.thomastriplet.net/) casually suggested that Domain Objects seemed to behave like interfaces between classes of people who know computers, and those who did not.

[11] [Fowler 2003] mentions "domain objects" in the DOMAIN MODEL chapter and elsewhere uses DomainObject as a LAYER SUPERTYPE in that context. [Larman 2004] describes the "Business Object Model" as a superset of the "Domain Model" artifact.

- Require DOMAIN OBJECTS to have versions
- Prescribe an approach to implementing DOMAIN OBJECTS.

4.2 Front Controller, Dispatchers and Commands

This Section presents a refinement of the FRONT CONTROLLER pattern described in [Fowler 2003]. FRONT COMMANDS are promoted out of the FRONT CONTROLLER pattern and split into DISPATCHERS, a pattern based on [J2EE]'s DISPATCH VIEW pattern, and COMMANDS. These additional patterns, and their interrelations, are also described in this Section.

4.2.1 Context

Entry into a WEA is an important step in any request. It is at this point that an application can prepare the request so that it may be processed consistently, and where the intent of the request is made clear. As discussed in Section 3.8, the use of the FRONT CONTROLLER pattern implies a separation between the entry point and that part of the application which processes a given request. Once at that part of the application, a decision is often required to select which VIEW should be reached.

4.2.2 Problem

The FRONT CONTROLLER pattern described in [Fowler 2003] acts as a CONTROLLER in a high-level application manner (the handler [Fowler 2003, p.346]) and in a use case CONTROLLER manner (the command [Fowler 2003, p.346])[12]. In fact, the short definition of a FRONT CONTROLLER that we quoted in Chapter 3 only describes the handler aspect: "A controller that handles all requests for a Web site"[Fowler 2003, p344].

The scope of a FRONT COMMAND, acting as a use case CONTROLLER, has not been well defined in terms of implementation. The implementation must clearly meet the requirements for that particular type of request (e.g. a successful login request must lead to the user being logged in), but there is no explicit description of what that means in terms of separation of concerns.

The concept of a FRONT COMMAND lacks cohesion, in that implemented FRONT COMMANDS both process requests, and make decisions about how requests should be processed and to which VIEWS they should be redirected. Or more simply, they both "Do" and "Decide", where the former requires visibility on the Domain layer, and the latter requires visibility on the Application layer.

4.2.3 Solution

We have adapted [Fowler 2003]'s definition of the FRONT CONTROLLER pattern to only describe an entry point to an application whose main responsibility is to decide the high-level course of a request ([Fowler 2003]'s "handler"), thereby acting as an application CONTROLLER, bringing the FRONT CONTROLLER in line with its initial description ([Fowler 2003, p.344). The remaining responsibilities assigned to [Fowler 2003]'s FRONT COMMANDS then fit into the DISPATCHER and COMMAND patterns.

Each request will follow a scenario of a use case, and a DISPATCHER implemented while following a use case in this manner takes on a very specific form. For a high-level use case, each line—or set of lines—representing something the system does can be represented by a corresponding line of code, the execution of a COMMAND by the DISPATCHER. The alternate—or exceptional—scenarios will be treated in the same fashion within a given DISPATCHER (Figure 4-2, lines 08-10). We describe this as the DISPATCHER dispatching to COMMANDS and VIEWS as dictated by a use case (Figure 4-2, line 06).

[12] Larman04 breaks the Controller GRASP pattern into two flavors, a façade controller representing the overall system or subsystem, and a use case scenario.

```
01 public class LoginDispatcher extends Dispatcher {
02
03 @Override
04 public void execute() throws ServletException, IOException {
05  try {
06   new LoginCommand(myHelper).execute();
07   forward("/WEB-INF/JSP/html/main_menu.jsp");
08  } catch (Exception e) {
09   forward("/WEB-INF/JSP/html/login.jsp");
10  }
11 }
12 }
```

Figure 4-2 A simple Login Dispatcher using SoenEA

```
01 public class LoginCommand extends Command {
02
03 ...
04
05 @Override
06 public void execute()
07  throws CommandException {
08  String username = helper.getString("username");
09  String password = helper.getString("password");
10
11  try {
12   helper.setSessionAttribute("CurrentUser",
13    UserInputMapper.find(username, password));
14  } catch (SQLException e) {
15   throw new CommandException(e);
16  } catch (MapperException e) {
17   getNotifications().add("Sorry, no user for that " +
18   "username and password combo.");
19   throw new CommandException("Sorry, no user for that " +
20   "username and password combo.");
21  }
22 }
23 }
```

Figure 4-3 A simple Login Command using SoenEA

At then end of each scenario in a use case, either the system gives feedback to the user, which is represented by dispatching to an appropriate VIEW (Figure 4-2, line 07), or it redirects to another use case, or part of a use case (Figure 4-2, line 09, is effectively taking the user back to the beginning of the Login use case).

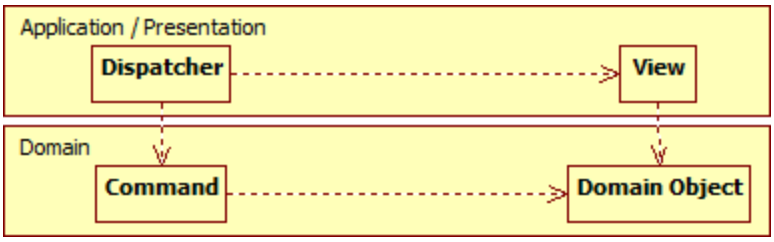


Figure 4-4 Separation between Dispatcher, View, Command and Domain Object Patterns

Promoting the "dispatcher" component from [Alur 2001]'s DISPATCHER VIEW pattern to a pattern in its own right, representing a solution to the problem of deciding what to do and what to show, we can consider DISPATCHERS as "deciding", and COMMANDS as "doing". The COMMAND then fits into the Domain layer, working primarily with DOMAIN OBJECTS, and has no dependency on VIEWS, which are in the layer above. Similarly separated, the DISPATCHER does not depend on DOMAIN OBJECTS, but does depend on VIEWS, COMMANDS, and even other DISPATCHERS (Figure 4-4).

4.2.4 Related work and contribution

Related work:

- [Fowler 2003] identifies the FRONT CONTROLLER pattern.
- [Fowler 2003] identifies FRONT COMMANDS as part of the FRONT CONTROLLER pattern.
- [Fowler 2003] identifies the LAZY LOAD pattern.
- [Fowler 2003] suggests that the FRONT CONTROLLER is a good place to implement entry point–specific features.
- The COMMAND pattern is a well established behavioral Gang of Four (GoF) pattern.
- [Alur 2001] identifies the DISPATCHER VIEW pattern.
- [Alur 2001] mentions the "dispatcher component" of the DISPATCHER VIEW pattern.

Our contribution:

- Using DISPATCHERS and COMMANDS to replace FRONT COMMANDS in the description of FRONT CONTROLLER
- Making an association between use cases and DISPATCHERS
- Drawing out a DISPATCHER pattern from [Alur 2001]'s DISPATCHER VIEW
- Identifying how DISPATCHERS would dispatch to COMMANDS and VIEWS

4.3 Lazy Load: Domain Object Proxy and List Proxy

This Section presents two approaches to LAZY LOAD:

- a DOMAIN OBJECT PROXY which provides a placeholder for a single DOMAIN OBJECT and
- a LIST PROXY which provides a placeholder representing a placeholder for many DOMAIN OBJECTS at once.

The Section also presents how the implementation of these approaches provides a systematic treatment of various problems associated with loading DOMAIN OBJECTS.

4.3.1 Context

[Fowler 2003] identifies a problem of performance loss due to the loading of huge numbers of interrelated objects. Fowler proposes interrupting such large loads by "leaving a marker in the object structure so that if the data is needed it can be loaded only when it is used" [Fowler 2003, p. 200]. Fowler then goes on to explain four main implementations of LAZY LOAD. Of the approaches Fowler describes, VIRTUAL PROXY is a good match for the DOMAIN OBJECT PROXY we promote and the VIRTUAL LIST PROXY would correspond to our LIST PROXY. What is emphasized in our description of these patterns is that they deal with DOMAIN OBJECTS.

4.3.2 Problem

A major problem that arises with DOMAIN OBJECTS is that they tend to be interconnected. Imagine if a Person class was defined as having Parents and Children. Even in a genealogical application it might prove cumbersome to load the entire data source into memory just to look at a single Person.

Given the simple example where a Person class stores a Person's name and identifies who their buddy is, the obvious case where one can run into trouble is when two People— for example Alice and Bob—are each other's buddy.

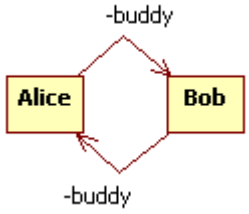


Figure 4-5 An Object Diagram showing two instances related in both directions by the role buddy

This example was used once before in Section 3.9. To reiterate, the problem is the implementation issue whereby loading up Alice causes the loading of Bob which causes the loading of Alice, etc. As mentioned previously, this seems simplistic and easy to avoid, but a cycle can be created from an arbitrary number of instances, and may not be at all obvious. Since cycles occur naturally in some domains, it is up to the application to ensure that they are handled correctly.

4.3.3 Solution

In terms of implementation, one might be able to come up with several solutions. One of the simplest that matches conceptually with how people think about such problems is to load the object of immediate concern and no objects beyond it. In circumstances where you can consider that there is a related object (that a person has parents), programmatically, one has a DOMAIN OBJECT PROXY (or PROXY LIST). This functions as a placeholder that is used superficially as a DOMAIN OBJECT (or List of DOMAIN OBJECTS), but which does not actually load anything from persistent storage, yet identifies that there is something to load.

Loading only a single DOMAIN OBJECT and not any DOMAIN OBJECTS that are its fields is also an implicit solution to the problem of cyclic references causing infinite loops of loading. Given the example in Figure 4-5, if an instance of Alice is loaded, only a placeholder for the instance of Bob will be loaded, and thus there is no cycle.

A PROXY should contain a field with the same value as the IDENTITY FIELD of the DOMAIN OBJECT that it represents, which will allow the PROXY to load its DOMAIN OBJECT when needed. An important part of using a PROXY with DOMAIN OBJECTS is that they should be treated as the same externally; equality and `hashcode` methods, depending on the language used, should be overridden so that DOMAIN OBJECTS and their PROXYS are treated accordingly. Given that a PROXY has the IDENTITY FIELD value available, checks for equality do not need to load the actual DOMAIN OBJECT.

LIST PROXYS need not concern themselves with equality. However, LIST PROXYS still need a means to load their content, and this is done by storing the containing DOMAIN OBJECT. For example, if a Person, Bob, has a List of buddies, a LIST PROXY representing that List would contain a Person field whose value was Bob.

Our approach is to always use a PROXY whenever another DOMAIN OBJECT is a field for a DOMAIN OBJECT that is being loaded. If an attempted load primarily involves loading multiple DOMAIN OBJECTS of the same type (as in the loading of the content of a LIST PROXY, `findAllBuddies(…)`), we propose that all objects created be DOMAIN OBJECT PROXYS. Whenever a DOMAIN OBJECT field represents a Collection of DOMAIN OBJECTS, we propose using a LIST PROXY (or some other form of COLLECTION PROXY)

What this leaves is a system where any request to load a DOMAIN OBJECT from the data source will never load more than a single DOMAIN OBJECT. It also greatly reduces the complexity of any given attempt to load a DOMAIN OBJECT. The tradeoff, as Fowler suggests, is that such systems may have to make more individual loads.

We do realize that this approach does not scale in many cases, in that more complex systems will eventually need optimization that may be inconsistent with our prescribed use of PROXYs, e.g. when loading the data for many DOMAIN OBJECTs at once. However, as a first pass to any development, it is a consistent and easily followed approach. Once a system is further developed (and has a comprehensive test suite), performance testing can identify where the use of PROXYs can be phased out as excessive, improving performance as needed.

4.3.4 Related work and contribution

Related work:

- [Fowler 2003] identifies the "Lazy Load" pattern.
- [Fowler 2003] associates "Lazy Load" with improving performance.
- [Fowler 2003] describes several implementations of "Lazy Load":
  - Lazy Initialization
  - Virtual Proxy / Virtual List
  - Value Holder
  - Ghosts
- [Fowler 2003] identifies the "ripple loading" problem that can stem from LAZY LOAD..

Our contribution:

- Explicitly associating LAZY LOAD with DOMAIN OBJECTS
- Identifying how LAZY LOAD reduces representational gap for developers
- Identifying cyclic references as an additional problem dealt with by LAZY LOAD

4.4 Identity Map

This Section presents a refinement of [Fowler 2003]'s IDENTITY MAP pattern, clarifying its scope in an application and identifying how it relates to DOMAIN OBJECTS. This Section also presents how the IDENTITY MAP pattern contributes to resolving previously discussed problems, and associates this pattern with the LAZY LOAD pattern.

4.4.1 Context

[Fowler 2003] describes the problem of potentially loading data from the "same database record into two different objects" [Fowler 2003, p.195], then potentially changing both of them independently and trying to coordinate saving that back to the database. Fowler's proposed solution is to use a form of map, relative to the current session, to store every object that gets loaded.

[Fowler 2003] discusses various implementation issues:

- "Choice of Keys"
- Use of either "Explicit or Generic" IDENTITY MAPS
- Correspondence between IDENTITY MAPS and classes
- Location of an IDENTITY MAP in the design

The IDENTITY MAP embodies the idea that all attempts to access a particular DOMAIN OBJECT that exists in persistent storage, access a single instance of that DOMAIN OBJECT in memory. We consider these accesses to be within a context (e.g. all accesses within a single thread). [Fowler 2003] discusses IDENTITY MAP and offers some suggested implementation guidelines.

4.4.2 Problem

Expanding slightly on [Fowler 2003], consider the problem of using sessions as the context for IDENTITY MAPs. In a WEA, a user may have multiple windows open, and may make multiple requests within the same session. Once again, sharing an IDENTITY MAP inside a session becomes a concurrency problem. In addition, there is often no explicit end to a session, save a server timing it out. Therefore anything recorded in a session can persist for an indeterminate amount of time, and an IDENTITY MAP may store a large number of DOMAIN OBJECTs. Lastly, within a given session, DOMAIN OBJECTs may be changed by users in other sessions. When such external changes happen, either any request initiating a change must check if there are any IDENTITY MAPs that hold that object and synchronize with any found—a daunting task, related to the "lost update" concurrency problem—or IDENTITY MAPs can become a source of "inconsistent reads" as they read in new data that is synchronized with the external state while maintaining older, now incorrect data.

Considering non-session related issues, we know that in WEAs, each request may have complex business logic, sometimes split into multiple COMMANDS. A difficult problem to detect arises when two different instances of the same DOMAIN OBJECT are loaded, then changed, as might happen in the case of multiple COMMANDs. Safely re-synchronizing the DOMAIN OBJECT with the data source can be difficult when such a dual loading/changing happens. Taken alone, our approach to LAZY LOAD described in Section 4.3.3 actually increases the likelihood of the same DOMAIN OBJECT being instantiated multiple times within the same request.

Given that we propose using PROXYS in conjunction with IDENTITY MAPs, we must decide whether only DOMAIN OBJECTs are stored in the IDENTITY MAP, or whether both DOMAIN OBJECTs and PROXYS are stored. We must also indicate when an IDENTITY MAP should be checked.

4.4.3 Solution

[Fowler 2003 p.198] suggests that an IDENTITY MAP avoids conflicts within a single session. This principle is true where a session is either explicitly serialized or runs in a single thread, which implicitly serializes the session. In modern WEAs, this serialization is not the case. Multiple concurrent requests can happen within a single session, which leads to the solution of associating IDENTITY MAPs with a single request, indirectly stating that an IDENTITY MAP exists for a fixed duration within the context of a thread. This solution eliminates all the problems of using the session context at the cost of having to rebuild IDENTITY MAPs for each request.

Within each request, the use of an IDENTITY MAP eliminates the problem of duplicate DOMAIN OBJECTs being created during a load, first by delaying the creation of additional DOMAIN OBJECT instances by using PROXYS (to prevent infinite loops), and then by checking against the IDENTITY MAP whenever a PROXY attempts to load the DOMAIN OBJECT that it represents. This completely avoids all the problems of trying to synchronize two instances of the same DOMAIN OBJECT within the same request.

Given the effective pairing of IDENTITY MAP with PROXYs, we feel that IDENTITY MAP should be explicitly stated as part of the LAZY LOAD pattern. The examples in [Fowler 2003] show the LAZY LOAD implementation accessing IDENTITY MAPs directly. The approach we favor has IDENTITY MAPs being accessed by the mechanism that does the actual loading (the MAPPER). Both approaches work well, but our approach slightly reduces coupling as MAPPERS will already have a dependency on IDENTITY MAPs.

To provide a consistent approach, we consider two questions:

- Where should an IDENTITY MAP get checked?
- Should an IDENTITY MAP contain PROXYs and DOMAIN OBJECTs or only DOMAIN OBJECTs?

As stated above, we favor checks to the IDENTITY MAP being made from the MAPPER instead of the PROXY. The reasoning is that PROXYs will access the same find methods in a MAPPER to instantiate their DOMAIN OBJECTs, as COMMANDs will use to instantiate any DOMAIN OBJECTs they need. If the MAPPER is responsible for finding these DOMAIN OBJECTs, then it should also be responsible for checking all the places where they might be, such as the data source or the IDENTITY MAP.

The purpose of an IDENTITY MAP is to provide access to previously loaded DOMAIN OBJECTs, and through its use, prevent their duplicate loading. As well, the principal benefit of storing proxies in the IDENTITY MAP would be eliminating the instantiation of PROXYs, which is not an intensive activity in that no database access is involved.

While [Fowler 2003] does not propose storing PROXYS in IDENTITY MAPs, we find that students regularly try to do so on the grounds that they do not wish to create unneeded PROXYs. The downside of this practice is that it makes the IDENTITY MAP more complicated, and adds another layer of checking wherever a PROXY might be instantiated—which is code that developers will work with often, in our experience. The minimal gains do not justify the extra complication, and as such we promote storing only DOMAIN OBJECTs in IDENTITY MAPs.

4.4.4 Related work and contribution

Related work:

- [Fowler 2003] identifies the IDENTITY MAP pattern.
- [Fowler 2003] identifies IDENTITY MAP as a means to improve correctness.
- [Fowler 2003] identifies IDENTITY MAP as a caching mechanism that can improve performance.
- [Fowler 2003] discusses some implementation issues with IDENTITY MAP.
- [Fowler 2003] associates concurrency management with IDENTITY MAP very briefly.

Our contribution:

- Proposing that IDENTITY MAP should be associated with a single request
- Recognizing how IDENTITY MAP contributes to our recommended solution of the cyclic reference problem mentioned in Section 4.2 (LAZY LOAD)
- Explicitly associating IDENTITY MAP with DOMAIN OBJECTS
- Describing when the IDENTITY MAP should be checked

4.5 Input Mapper and Output Mapper Patterns

This section presents a significant refinement of [Fowler 2003]'s DATA MAPPER pattern, identifying new patterns and incorporating optimistic concurrency management. This section also presents how these new patterns interact with DOMAIN OBJECTS, in particular giving guidance on initial optimization approaches.

4.5.1 Context

In order to have DOMAIN OBJECTS, we need data from some source. Barring storage in an Object-Oriented Database, this data is usually stored as primitive types. Getting to and from this primitive state needs to be done carefully to ensure smooth working of a system.

4.5.2 Problem

The DATA MAPPER pattern has two distinct responsibilities, getting data from, and sending it back to, the data source. Besides the fact that DOMAIN OBJECTS and their data source are common participants for both behaviors, the processes involved for transfers in either direction is completely independent—yet they are identified together in the same pattern. While [Fowler 2003] suggests the possibility of splitting out Finders into a SEPARATED INTERFACE, which would partially address this problem, there is nothing specifically mentioned about splitting out direct access to the data source, which is an orthogonal problem.

4.5.3 Solution

Fowler's short definition of a Data Mapper is

"A layer of Mappers (473) that moves data between objects and a database while keeping them independent of each other and the mapper itself" [Fowler 2003, p165]

To expatiate on the identified goals, we propose a definition of what could be called DOMAIN MAPPERS:

classes that map between persistent storage and in-memory DOMAIN OBJECTS, both data and structure that are necessary to keep the DOMAIN OBJECTS consistent, while maintaining separation of concerns.

Fowler discusses using a SEPARATED INTERFACE [Fowler 2003 pg.176] to move the implementation of the "find" methods outside of the DATA MAPPER. Building upon this idea we split the entire "find" behavior away from the DATA MAPPER, and are left with two flavors of DATA MAPPER, the INPUT MAPPER and the OUTPUT MAPPER. The INPUT MAPPER corresponds to the external behavior described by Fowler's FINDERS, but neither MAPPER is dependant on the other.

One of the most important differences is that all the behavior that Fowler identified could be factored out of the DATA MAPPER is, in our system, identified as being completely cohesive. Additionally, the remaining behavior in the DATA MAPPER is also cohesive and there is no coupling between these components. The INPUT MAPPER will then "input" data to instances of DOMAIN OBJECTS and the OUTPUT MAPPER will "output" data from instances of DOMAIN OBJECTS to the data source.

A further split is to remove all direct data source access from INPUT and OUTPUT MAPPERS and place them in FINDERS and TABLE DATA GATEWAYS (TDGs), respectively. For an SQL database, Finders would contain all the select statements and TDGs would have the standard update/insert/delete SQL as well as any other data modification statements.

The original definition considered only databases. From an abstract sense, the term 'database' and 'persistent storage' are interchangeable. 'Database' is also an overloaded term in this domain, often understood to be a service like a MySQL or Oracle server. The term persistent storage allows the definition to cover xml or other flat-file systems, as well as any other means of persisting data that can be produced. As with the original DATA MAPPER pattern (and related patterns), it is strongly advised that the persistent storage mechanism used be ACID-compliant to maintain reliable behavior.

A point made in [Fowler 2003]'s section on the DATA MAPPER pattern is that the goal is "to minimize database queries". This is tempered by Fowler's regular advice to do it correctly first and optimize later. Our position is that, particularly in the case of INPUT/OUTPUT MAPPERS, early iterations should focus more on making the INPUT/OUTPUT MAPPERS as simple as possible, regardless of the number of database queries.

An initial approach to an OUTPUT MAPPER design can be demonstrated with an example of a delete method :

```
01 public void delete(Person d) throws SQLException, MapperException,
02  LostUpdateException{
03  int count = PersonTDG.delete(d.getId(), d.getVersion());
04  if(count==0) throw new LostUpdateException();
05  PersonTDG.deleteBuddyRelationWithPersonId(d.getId());
06  PersonTDG.deleteBuddyRelationWithBuddyId(d.getId());
07 }
```

Figure 4-6 An example OutputMapper delete method

Cascading deletions are explicitly shown in the `delete` method instead of being hidden in the database. Either explicit calls to delete multiple objects or some UNIT OF WORK mechanism would make a call for each `Person` to be deleted. Thus the method bodies of OUTPUT MAPPERS number often less than 10 LOC.

Also note that an OUTPUT MAPPER supports optimistic concurrency management by checking for lost updates and reporting them. In conjunction with TABLE DATA GATEWAYS, this provides an effective means of detecting this form of concurrency problem.

The advantage is in simple and clear code, as in Figure 4-6. If a single `delete` call were made to `PersonTDG`, then the responsibility of figuring out what it means to delete a `Person` would be relegated to the GATEWAY. If cascades were done in the database, then that information would not exist in the code. Our suggestion is to follow the patterns simply first, before considering optimization, and we explicitly state that as a heuristic of the INPUT/OUTPUT MAPPER patterns in particular.

The question of how much data to pull back from persistent storage in one request is related to optimization. Optimization is an effort that should be applied once a system is put together and metrics can be gathered as to where optimization will do the most good. Requests to an INPUT MAPPER can be used to create either a particular

DOMAIN OBJECT, or List of DOMAIN OBJECTs. At most, only those requested DOMAIN OBJECTs should be created, with all related DOMAIN OBJECTs–be they fields in the specified DOMAIN OBJECT or Lists of DOMAIN OBJECTs that represent some sort of aggregate associated with the DOMAIN OBJECT in question–attached via a PROXY (or PROXY LIST).

The primary use of this LAZY LOAD approach is to prevent cyclic reference infinite loops. There are often optimization benefits, but there is also associated overhead that should be a consideration once the initial phases of development are complete and streamlining needs to begin. If one can guarantee no cyclic references and that large lists of PROXYs—that all get used—are being generated then it is often wise to forego the LAZY LOAD and instantiate the DOMAIN OBJECTs directly. This is carried out only when there are metrics which indicate the need for that variety of optimization.

The OUTPUT MAPPER not only takes the data from the DOMAIN OBJECTs passing it through to the TDG, it must also represent the structure of the overall DOMAIN MODEL. While it is easy to associate a DOMAIN OBJECT to a corresponding row in a database table, there is often data found in other tables that will be affected by changes in a DOMAIN OBJECT. These secondary effects are often described as either cascades, or demonstrations of aggregation or composition.

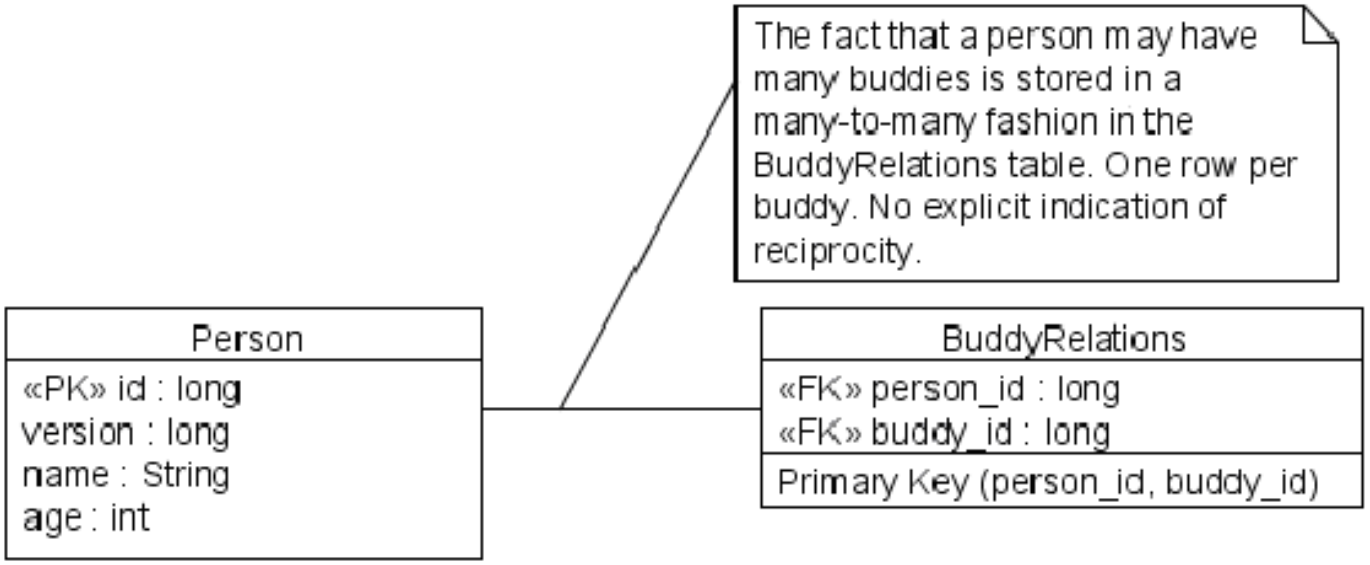


Figure 4-7 Output Mappers store domain logic regarding object relations

Person

id	version	name	age
1	1	Bob	21
2	1	Alice	22
3	1	Dave	24
4	1	Eve	24

BuddyRelations

person_id	buddy_id
1	3
1	2
1	4
2	3
3	1

Figure 4-8 example of tables for a Person described in Figure 4-7

When considering the deletion of "Bob", the three different shadings in Figure 4-8 represent the explicit removal in the Person table, and the two varieties of implicit side effect (in BuddyRelations) that we would like to make explicit in the PersonOutputMapper.

It is the OUTPUT MAPPER's representation of the structure that allows the decision of what to delete to be made explicit in the PersonOutputMapper's delete method (Figure 4-6). While a PersonTDG would explicitly define the methods that would communicate with the database to effect the actual changes, PersonOutputMapper's `delete` method would specify that deleting a Person means removing that Person from persistent storage (the red/solid shading in Person, Figure 4-8). It would also mean removing all BuddyRelations where that Person is the subject of the BuddyRelation (green/brick shading, three rows in BuddyRelations, Figure 4-8), and removing all BuddyRelations where that Person is the object of the BuddyRelation (blue/dithered shading in BuddyRelations, Figure 4-8)

Cascades are a separate concept from aggregations and compositions. Cascades represent the logic of what happens to other DOMAIN OBJECTS when a related DOMAIN OBJECT is affected. This logic is generally directly in the database, but is very much domain logic, hence we deal with it in the Domain layer instead of leaving it to the Technical Services layer or below.

Aggregations and compositions describe how DOMAIN OBJECTS can be related. Simple associations are also used to relate DOMAIN OBJECTS, but their consideration is not problematic and is not dealt with directly in the INPUT/OUTPUT MAPPERS except where cascades are concerned.

One of the common features of frameworks like Struts [Struts] and Hibernate [Hibernate] are some facility for dealing with the one-to-one, many-to-one and many-to-many relationships that give a relational database structure in terms of a DOMAIN MODEL. Presuming a well normalized database (at least 3NF or BCNF), duplication is minimized, and parallels can be drawn between DOMAIN OBJECTS and the database. This is true regardless of framework.

Our approach differs from Hibernate in the location where one manages these relationships. One-to-one relationships are handled entirely by the UNIT OF WORK, which is standard. They are represented as foreign key fields in a database table that already represents a DOMAIN OBJECT. The preference is not enforce this with database mechanisms. If two DOMAIN OBJECTS become related, the containing object will be registered *dirty* when the other object becomes contained by it. On update, the foreign key will be set appropriately, removing any previous relationship. In Hibernate, such relations are represented in Hibernate's configuration files, and as such the underlying mechanism is hidden.

Many-to-one, or the general case of many-to-many, can be considered as having two approaches, however the first uses the UNIT OF WORK approach described above and represents composition. More often, such relationships are independent of the identity of either related DomainObjects. As such, the record of those relations is kept independently of the associated DomainObjects. In this second situation, updating the containing object generally involves deleting all of the previous relationships and then creating the new ones[13].

Figure 4-9 Two tables, representing a one-to-one relationship

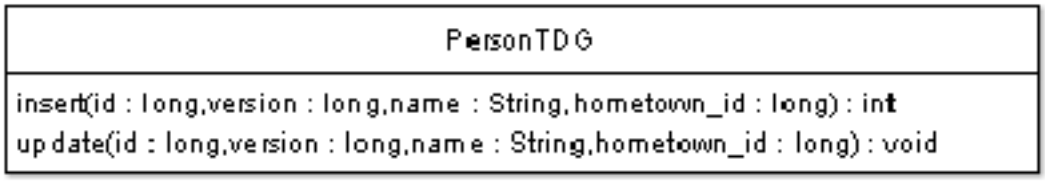


Figure 4-10 The concrete TDG for the Person Domain Object

Consider a function allowing a person to record their home town. If this were to be represented as a one-to-one relationship, the Person table might have a foreign key, `hometown_id` (Figure 4-9). A DomainObject in memory of type Person would have an attribute of type Town. Upon the initial creation of a Person, the PersonOutputMapper would send values for `id`, `version`, `name` and `hometown_id` to the TDG's `insert` method. As far as the PersonOutputMapper is concerned, the Town should already exist, therefore only the `id` of that Town needs to be known; where that comes from is not the concern of the PersonOutputMapper.

[13] Of course, optimization concerns can lead to variation. It is possible that removing all associations and creating them anew could be costly. However, that logic could be worked into the OutputMapper when it was determined to be appropriate, and with the possible exception of the TDG, all other classes would be oblivious to this concern.

Similarly, if a Person's hometown changed, the update method would be called and the Town table would still not be affected.

Imagine a Person "Stuart" with id 1 and two Towns, "Montreal" and "Huntingdon", with ids 3 and 4, respectively. Upon initial creation of the system, imagine that the Towns were added with an initialization script. The first Person added, "Stuart", might be associated with the Town "Montreal". The PersonOutputMapper would be responsible for dealing with the insert request for Person "Stuart" (as the newly created DOMAIN OBJECT would be registered *new*), and would call the insert method in the PersonTDG using the call in Figure 4-11.

```
insert(myPerson.getId(), myPerson.getVersion(), myPerson.getName(),
myPerson.getHomeTown().getId());
```
or with literals:
```
insert(1, 0, "Stuart", 3);
```

Figure 4-11 inserting a person

Later, it is learned that "Stuart", currently lives in "Montreal" but actually is from "Huntingdon". The application would be used to update the Person "Stuart", who would then be registered *dirty* prompting the PersonOutputMapper to deal with the update request for Person "Stuart". This would prompt a call to the update method in the PersonTDG (Figure 4-12).

```
update(myPerson.getId(), myPerson.getVersion(), myPerson.getName(),
myPerson.getHomeTown().getId());
```
or with literals:
```
update(1, 1, "Stuart", 4);
```

Figure 4-12 updating a Person

The important thing to note is that only the PersonOutputMapper and the PersonTDG were used. No Town DOMAIN OBJECT was changed.

4.5.4 Related work and contribution

Related work:

- [Fowler 2003] defines a DATA MAPPER.
- [Fowler 2003] describes DATA MAPPERs using a "rich constructor" ([Fowler 2003 p169]).
- [Fowler 2003] proposes LAZY LOAD can address a problem with "rich constructor", the cyclic load[14].
- [Fowler 2003] proposes that DATA MAPPERs should insert newly created objects into IDENTITY MAPs, and that doing so after using a blank constructor is an effective way to avoid cyclic loading.
- [Fowler 2003] describes splitting Finders out of the Data Mapper.

Our contribution:

- Splitting DATA MAPPER into INPUT MAPPER and OUTPUT MAPPER
- Splitting direct data source access out of the MAPPERS and into TABLE DATA GATEWAYS
- Explicitly including the evaluation of optimistic concurrency as a responsibility of OUTPUT MAPPERS
- Providing guidance on the degree of optimization to consider during initial development
- Providing guidance for how the MAPPERS interact with other patterns[15]

[14] Fowler suggests that the solution is "messy" ([Fowler 2003 p169])
[15] Covered in Sections 5.2.2 and 5.2.6

4.6 Table Data Gateway (TDG) and Finder

In this Section we present a refinement of the TDG pattern, separating it from the DATA MAPPER pattern in a fashion consistent with our description of the OUTPUT MAPPER pattern in Section 4.5. We also present a newly identified sub-pattern of TDG, the FINDER, which corresponds more closely to the INPUT MAPPER PATTERN. In this Section, both concurrency and security problems are also addressed.

4.6.1 Context

Abstracting software systems into layers is a common practice. A Services layer is where the more technical access to data is often found. Several patterns/structures are used to describe how this layer functions or what its primary components might be. [Fowler 2003] describes a few, some straddling the boundary to the Domain layer above: TABLE DATA GATEWAYS, TABLE MODULES, DATA MAPPERS, ROW DATA GATEWAYS, ACTIVE RECORDS and DATA TRANSFER OBJECTS, among others[16].

Without going into the details of these patterns, one can still see a general purpose. They represent a means to separate the details of accessing a data source implementation from the rest of an application's use of the data/objects. The various named patterns are not prescriptive. These patterns describe how this separation of concerns has been implemented in various ways.

4.6.2 Problem

Accessing data in a database

Once data is acquired it must be stored for later use, and thus is made persistent. In WEAs, this is primarily done with a database. Developers need a way of interacting with this database while being as oblivious as possible to the underlying data source. In considering how to accomplish this practically, several methods, such as the Service layer patterns mentioned above, have been proposed[17]. A main problem is choosing the correct approach from the many that exist.

Avoiding Lost Updates, Optimistic Concurrency Management

Lost Updates are a serious problem that can lead to incorrect data. A Lost Update occurs when the same Domain Object is updated in two different transactions at essentially the same time. The user who makes the first update thinks their changes are successfully persisted. The user making the second update may overwrite the first users change immediately afterwards, without knowing that they have done so.

Ensuring that data is sanitized

Security considerations are often overlooked or considered something to be dealt with later. Security is becoming more and more critical, particularly with the volume of monetary information and personal data stored in WEAs increasing. As such, security considerations should be explicitly addressed, and a TDG, being a vulnerable boundary between two systems, is a place that needs such consideration.

[16] Microsoft has DATA ACCESS OBJECTS, for example.
[17] There are many more. Even stored procedures in the database can be considered a part of this. Fowler acknowledges having seen stored procedures serving essentially as TDGs for an application (as have we).

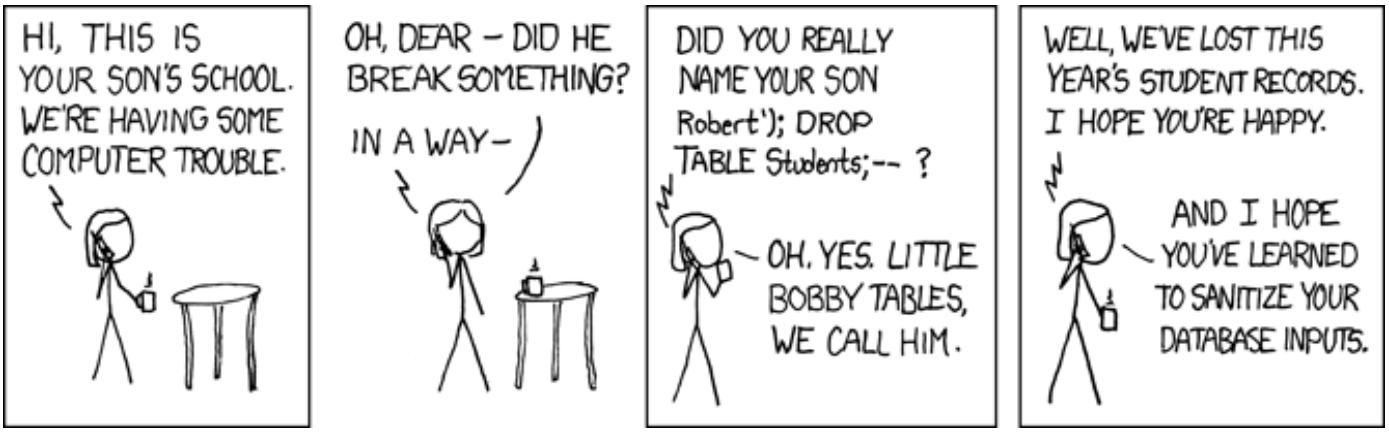


Figure 4-13 [XKCD, "Exploits of a Mom", http://xkcd.com/327/]

4.6.3 Solution

Accessing data in a database

The deciding factors are simplicity, separation of concerns and the showing of intent. Our choice of TDG, in combination with our OUTPUT MAPPER pattern, represents what we feel to be the optimal implementation in keeping with the four pillars of good design outlined by Kent Beck:

- It should be simple
- It should show intent
- It should meet user requirements
- It should be easily maintainable

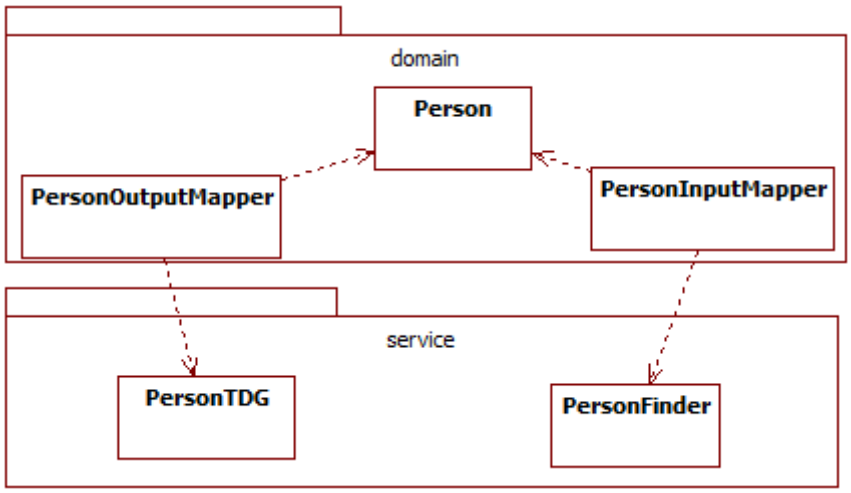


Figure 4-14 How DOs, I/O Mappers, TDGs and Finders relate

A TDG's methods take only primitive data types. This allows TDGs to avoid upwards dependencies and highlights the separation of concerns between them and their corresponding INPUT/OUTPUT MAPPERS. INPUT MAPPERS pass the parameters that will eventually fill out SELECT statements and OUTPUT MAPPERS communicate all primitive data that corresponds to each column that needs to be updated or inserted from the DOMAIN OBJECT.

Concerns are separated as follows[18]:

- (Domain Object) Providing an interface to the data that the rest of the application can use
- (Mappers) Providing an interface on the persistence mechanism for the data for the rest of the application[19]
- (InputMapper) Mapping the data to a useable object[20]
- (TDG) Changing the data source
- (TDG/Finder) Sanitizing the data
- (Finder) Reading data from the data source

The design fits in the layered scheme without straddling bounds, keeps high cohesion and fairly low coupling, and can be consistently applied across all DOMAIN OBJECTS.

Avoiding Lost Updates, Optimistic Concurrency Management

Lost updates are mentioned in Fowler, as is optimistic concurrency management[21]. A good solution is even given. However, this solution is not mentioned as explicitly belonging in a TDG. It is considered a separate pattern (OPTIMISTIC OFFLINE LOCK(416) [Fowler 2003]). We go so far as to say that a TDG is wrong if it does not implement this protection.

```
01 private static final String UPDATE_STRING =
02  "UPDATE Person SET name = ?, age = ?, buddy_id = ?, " +
03  "version = (version + 1) " +
04  "WHERE id=? AND version=?;";
05
06 public static int update(long id, long version, String name,
07  int age, long buddy_id)
08  throws SQLException {
09  Connection con = DbRegistry.getDbConnection();
10  PreparedStatement ps = con.prepareStatement(UPDATE_STRING);
11  ps.setString(1, name);
12  ps.setInt(2, age);
13  ps.setLong(3, buddy_id);
14  ps.setLong(4, id);
15  ps.setLong(5, version);
16  int result = ps.executeUpdate();
17  ps.close();
18  return result;
19 }
```

Figure 4-15 update in a TDG

The TDG is responsible for determining from the database whether any changes were actually made. In Figure 4-15, if the result is 0, no rows were updated, and there was likely a lost update. A similar thing happens with delete methods in a TDG. We have seen alternate approaches where the update is made and then the version is checked. It all boils down to interacting with the data source to determine if versions correspond.

Ensuring that data is sanitized

[18] Concerns regarding "lost update" issues are not explicitly mentioned here. Each piece has a part to do. Domain Objects store the version, Mappers notify the world about it when there is a problem, Unit of Work passes the buck and either a Session Command or a Dispatcher will decide what to do about it. Even the UI can have a big part, giving a merge interface in a nicely done application. The TDG's contribution is strictly in how it interacts with the database.

[19] The provision of interfaces implies as intuitive an Object Oriented interface as possible for both the data and the persistence mechanism.

[20] In some implementations, data is not mapped beyond what is returned from the database driver, something like a RecordSet. This is sometimes used explicitly, or wrapped in an interface. Some implementations do (The Data Mapper's primary purpose)

[21] "Optimistic Offline Lock" is the variation mentioned in [Fowler 2003], a term that is perhaps misleading in that database locking is not used.

While this is strictly an implementation issue, it has become apparent that data sanitizing is infrequently applied. Many languages offer simple solutions (e.g. Java's `PreparedStatement`). When such solutions do not exist, they should be implemented. The security risk associated with this is such that it is architecturally relevant.

More specifically, data coming down to the TDG invariably comes in from the user interface. Either by accident, or through intent, unsanitized data can lead to trouble, such as maliciously crafted data that subverts SQL statements. As the trouble happens to the database, GRASP suggests that the responsibility for addressing it lies as close to the database as possible. Fortunately, most modern languages/drivers subscribe to this view and have easy mechanisms for sanitizing data. Unfortunately, our experience has shown that many programmers still build their SQL with String concatenation or some form of `printf`[22].

4.6.4 Related work and contribution

Related work:

- [Fowler 2003] provides a pattern definition for TDGs.
- [Larman] suggests TDGs are a good place to isolate SQL away from the Mapper.
- [Fowler 2003] mentions using TDGs with Mappers when you "prefer handcoding for the actual mapping to the domain objects" [P.146][23].
- [Fowler 2003] also describes the general Gateway pattern.
- [Fowler 2003] hints that one could have a separate TDG for views and "interesting queries"[Fowler 2003, p145].
- [Fowler 2003] gives an implementation of optimistic concurrency management for lost updates[24].

Our Contribution:

- Pairing Mappers with Gateways as the primary means retrieving and storing Domain Objects.
- Explicitly including optimistic concurrency management in TDGs
- Explicitly stating that data inputs should be sanitized
- Promoting of the Finder pattern to isolate the SELECT statements from the TDG's data modification statements

[22] In PHP, for example, developers can still use printf while sanitizing their parameters using methods like `mysql_real_escape_string()` on the parameters.

[23] From our perspective, this is not so much about hand-coding, but about code generation / reflexive programming being from an OO perspective instead of from a data source perspective. Fowler does lots of neat things to make for less code via reflection, but this often proves to be less simple and shows less intent. In any EA of any complexity whatsoever, the mapping of data source to Domain Objects needs to be explicit and clear, as this is the key thing that ties in with other artifacts describing the structure of the database and system. At times students try to take the reflexive, data-centric approach to more complex systems… they start off looking like they are ahead (fast domain layer code generation, usually) until teammates stop understanding what is happening and the details of complex Domain Objects creep in, leaving difficult-to-trace bugs.

[24] Just not explicitly in the TDG

5 Applied and Improved Design: The SoenEA Framework and its Use

SoenEA is a framework we have developed over the last decade that encapsulates the best practices and guidance that we have acquired, particularly in the use of the patterns described in this thesis. The goal of SoenEA is to make it easier to write quality Web-based EAs. To this end, SoenEA includes implementations for many patterns described in this thesis, helpful utilities and sample code that serves both as examples of how to use SoenEA and as ready-to-use production-level code. We wrote SoenEA to

- help eliminate tedious tasks,
- help programmers to make fewer mistakes, and
- give guidance on proper practices.

While SoenEA was written with Java in mind and incorporates many Java-specific features, the patterns it embodies can be applied to other languages.

5.1 SoenEA, our WEA Framework

SoenEA can be split into four areas:

- Patterns
- Utility components
- Default Implementations of Typical Components (DITCs)
- Test components

The most important area is the patterns, which corresponds directly to the patterns discussed in this thesis. Partial or complete implementations of most of these patterns allow developers to quickly begin implementing their business logic while writing code that is consistent with our prescribed approach.

The utilities smooth out working in the web development environment. Some of these utilities may not be commonly found in other such frameworks. Other utilities, such as our DbRegistry, represent a very common implementation of access to a database, typical most web frameworks with which we have worked.

The DITCs consist of those classes that make use of the patterns and utilities to demonstrate how a developer might make use of SoenEA. These default implementations also provide some out-of-the-box resources that can be used directly in an application instead of having to re-implement them. The test components ensure that SoenEA is working properly after updates.

In summary, the patterns of SoenEA help to ensure that the code is right. Like a jigsaw puzzle, effort has been made to ensure it is difficult to assemble the pieces incorrectly. The utility components help make implementation easy, and often utility classes are hidden behind parts of a pattern, enabling their simple use. The DITCs facilitate the development of web applications, as they are ready-made pieces. Both the DITCs and the tests provide examples, giving further guidance on best practices for developing with SoenEA.

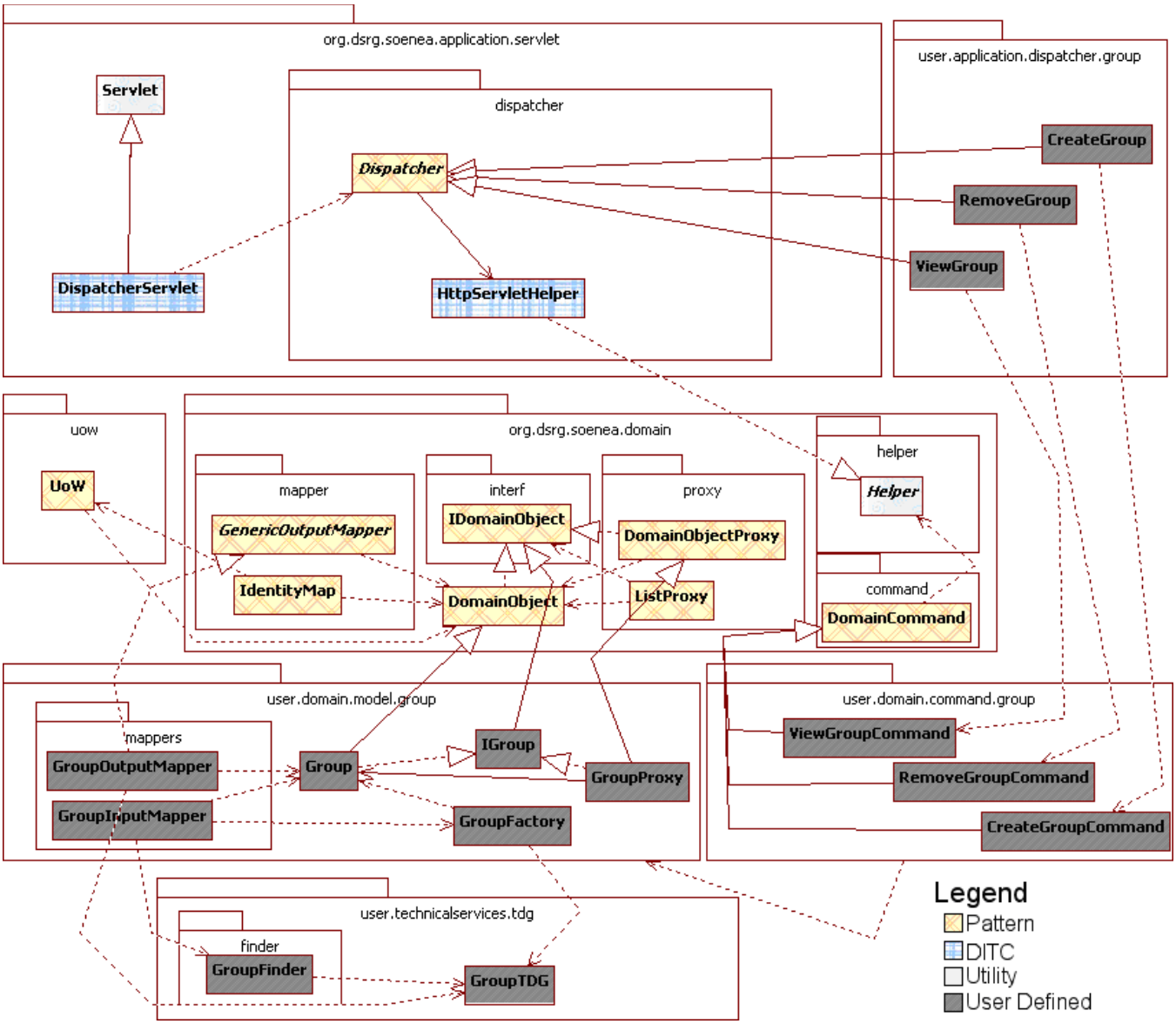


Figure 5-1 SoenEA general usage diagram

5.2 SoenEA Patterns

Figure 5-1 shows pattern classes from the SoenEA framework as well as sample user-defined classes. Such user-defined classes generally constitute the basic building blocks of a real application. In this section, we explain the thought process and activities of a developer while creating such classes.

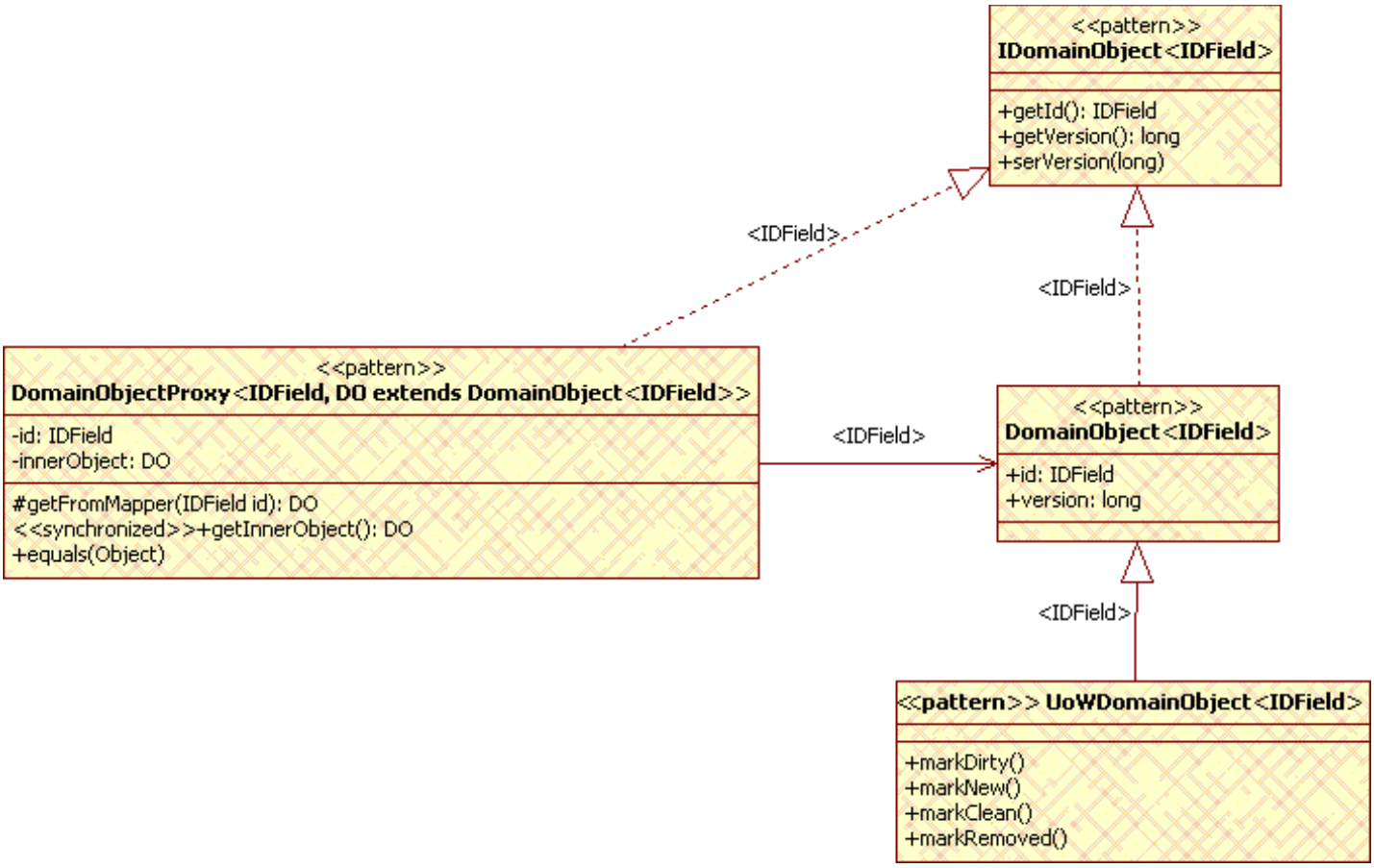


Figure 5-2 Domain Objects

5.2.1 Domain Objects

As described in Section 4.1, DomainObject<IDField>[25] instances have an `id` and a `version`. The interface IDomainObject provides get methods for both `version` and `id`, but only `version` can be set, as changing an `id` on a DomainObject would not be consistent with maintaining the identity of DomainObject instances. The DomainObjectProxy class acts as a generic proxy on DomainObjects, storing an `innerObject`[26] that is used in delegation, and an `id` that acts as a key to find the `innerObject` as it is needed. The DomainObjectProxy also provides the `getInnerObject`() method to aid in delegation and the abstract method `getFromMapper(…)` that is intended to be the means by which a DomainObjectProxy gets its `innerObject`. Both DomainObjectProxy and DomainObject have overridden `equals(…)` methods to allow tests that compared classes have the same id and are conceptually of the same type (e.g. IPerson and Person are both "people").

[25] The generic parameters such as <IDField> are programmatically useful. However, we feel that once it is made clear where they are, it is much easier to read diagrams and text if they are omitted. From time to time they will still be included where we feel they serve as a clarification or a reminder of their existence.

[26] In the Chapter describing Fowler's description of proxy, this field was referred to as realObject.

The `IDField` type parameter of DomainObject allows domain objects to have a variety of key types. The key should correspond to the primary key in the data source, usually a `Long`. The DomainObject-Proxy's second type parameter is a user-defined implementation of a DomainObject. This ensures that at compile time, when `getFromMapper()` is called, an appropriate `id` can be passed and an expected type can be returned.

DomainObject and UoWDomainObject represent two approaches for using UoW. The most basic approach is to not associate DomainObjects with the UoW at all. This is what Fowler dubs "caller registration": i.e., any time a Domain Object needs to be registered with UoW, the client (either a COMMAND or FACTORY) class is responsible for explicitly calling the appropriate UoW register methods, as illustrated in Figure 5-3 (lines 11 and 21).

```
1  public static Group createNew(String name,
2   List<IGroupMembership> members) throws
3   SQLException {
4    return createNew(GroupTDG.maxId(), 1, name, members);
5  }
6
7  public static Group createNew(Long id, long version,
8   String name, List<IGroupMembership> members) throws
9   SQLException {
10   Group result = new Group(id, version, name, members);
11   UoW.getCurrent().registerNew(result);
12   return result;
13 }
14
15 public static Group createClean(Long id,
16  long version, String name,
17  List<IGroupMembership> members)
18  throws SQLException {
19   Group result = new Group(id, version,
20    name, members);
21   UoW.getCurrent().registerClean(result);
22   return result;
23 }
```

Figure 5-3 GroupFactory Methods

In an alternative approach, called "object registration", DOMAIN OBJECTS manage their own UoW state. The UoW still has its same methods called, but the UoWDomainObject class provides methods that allow access to the UoW through the DomainObject itself (see Figure 5-2). Additionally, the constructor can base UoW status on the passed `version`/`id`, and setter methods in the user's subclass can explicitly call the `markDirty()` method so that the use of UoW can be transparent.

How a developer would use the Domain Object related patterns

Figure 5-4 shows how developer-implemented classes should subclass DomainObject to implement the DOMAIN OBJECT pattern. Developer-implemented interfaces for their DomainObjects should extend IDomainObject. Developers should create a PROXY by sub-classing DomainObjectProxy and implementing their DomainObject's interface. Their overridden `getFromMapper(IDField id)` method should call an appropriate InputMapper.

A FACTORY should be created for each DomainObject. The FACTORY contains at least one `createClean(…)` and one `createNew(…)` method, and each of these methods should make the appropriate calls to a UoW. In Figure 5-3, we show how two `createNew(…)` methods can be written to make the creation of new Groups more convenient. The parameters for these create methods, besides `id` and `version`, are the fields of Group, `name` and `groupMembership` (the IGroupMembership DOMAIN OBJECT not being shown here). It also demonstrates that providing new `id`s is the responsibility of the TDG.

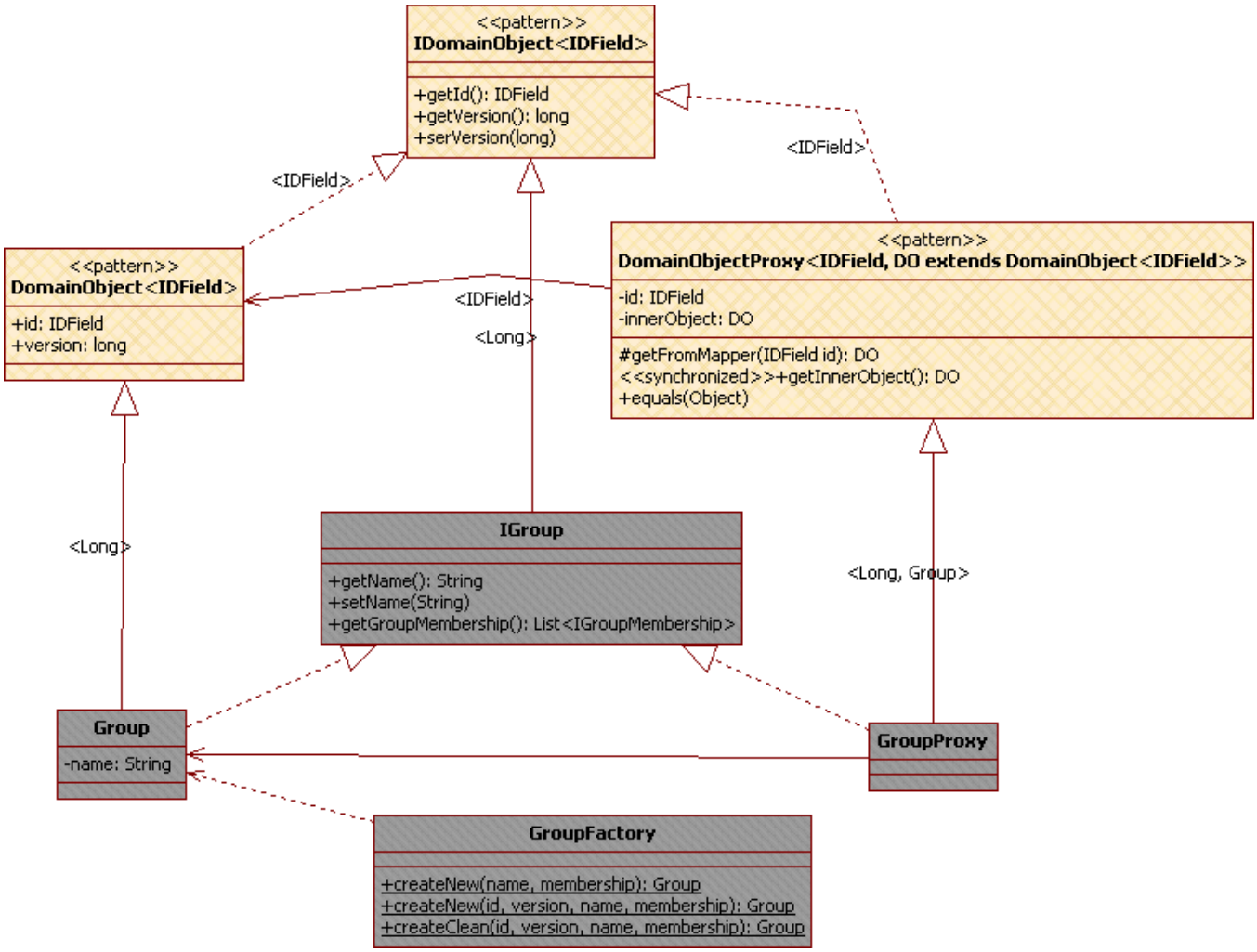


Figure 5-4Creating Domain Objects

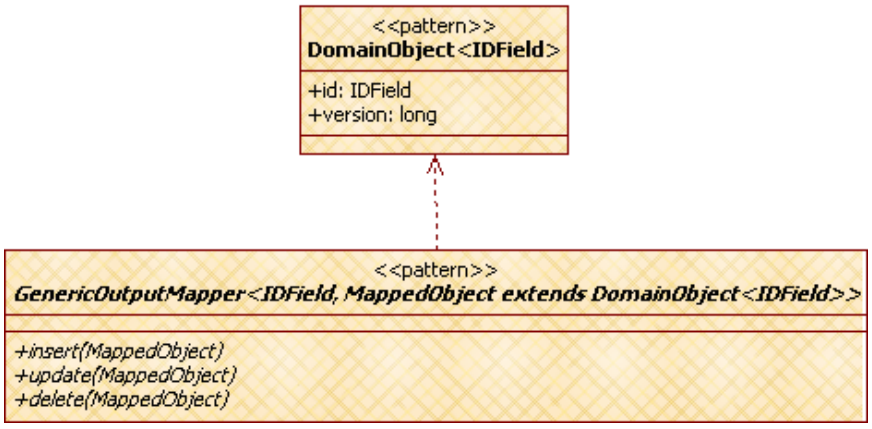


Figure 5-5 GenericOutputMapper

5.2.2 GenericOutputMapper

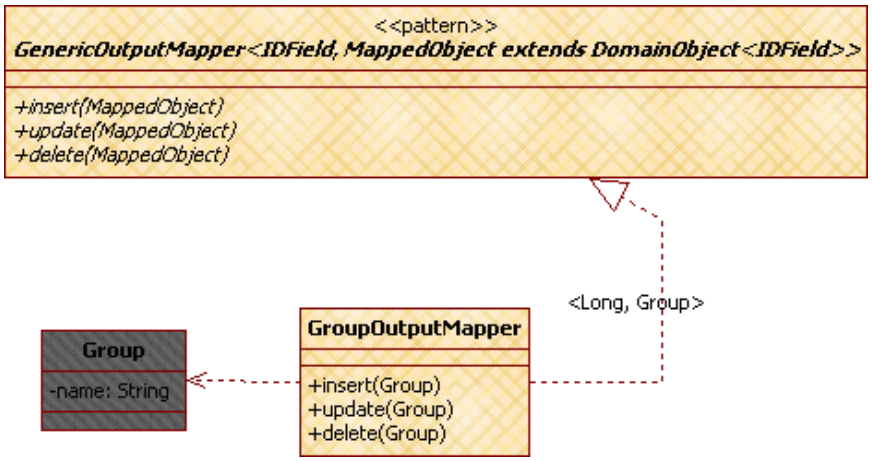


Figure 5-6 Creating a GenericOutputMapper

Figure 5-5 demonstrates the interface provided by GenericOutputMapper. The use of the parameterized type `MappedObject` is convenient in allowing IDEs to generate appropriate method headers, but is primarily used to enforce compile-time checking in other parts of SoenEA. The `insert`/`update`/`delete` methods provided in this interface correspond to the new/dirty/removed registries in that will be seen in section 5.2.3 on UoW.

How a developer would implement GenericOutputMapper

Once a user-defined DomainObject class is written, a GenericOutputMapper for that DomainObject can be created, as in Figure 5-6. The `insert()`, `delete()` and `update()` methods extract data from the passed DomainObject (`MappedObject`) and call appropriate TDG methods, passing the extracted data.

Figure 5-7 shows how the `update()` and `delete()` methods should check that the return value from the TDG is not zero (lines 11 and 21), as that generally indicates a lost update. There are several types of SQLException that can be detected and dealt with according to the user's needs, such as deadlock exceptions (a variety of lost update) or constraint failures on inserts. When any of these problems arise, a MapperException should be thrown.

```
1  public void insert(Group group) throws MapperException {
2   try {
3     GroupTDG.insert(group.getId(), group.getVersion(), group.getName());
4   } catch (SQLException e) {
5     throw new MapperException("Could not insert Group " + group.getId(),e);
6   }
7  }
8  public void update(Group group) throws MapperException {
9   try {
10    int count = GroupTDG.update(group.getId(), group.getVersion(), group.getName());
11    if(count == 0) throw new LostUpdateException("GroupTDG: id " + group.getId() + " version " + group.getVersion());
12    group.setVersion(group.getVersion()+1);
13  } catch (SQLException e) {
14    throw new MapperException("Could not update Group " + group.getId(),e);
16  }
17 }
18  public void delete(Profile object) throws MapperException {
19   try {
20     int count = GroupTDG.delete(group.getId(), group.getVersion());
21     if(count == 0) throw new LostUpdateException("GroupTDG: id " + group.getId() + " version " + group.getVersion());
22   } catch (SQLException e) {
23     throw new MapperException("Could not delete Group " + group.getId(),e);
24   }
25  }
```

Figure 5-7 GroupOutputMapper Methods

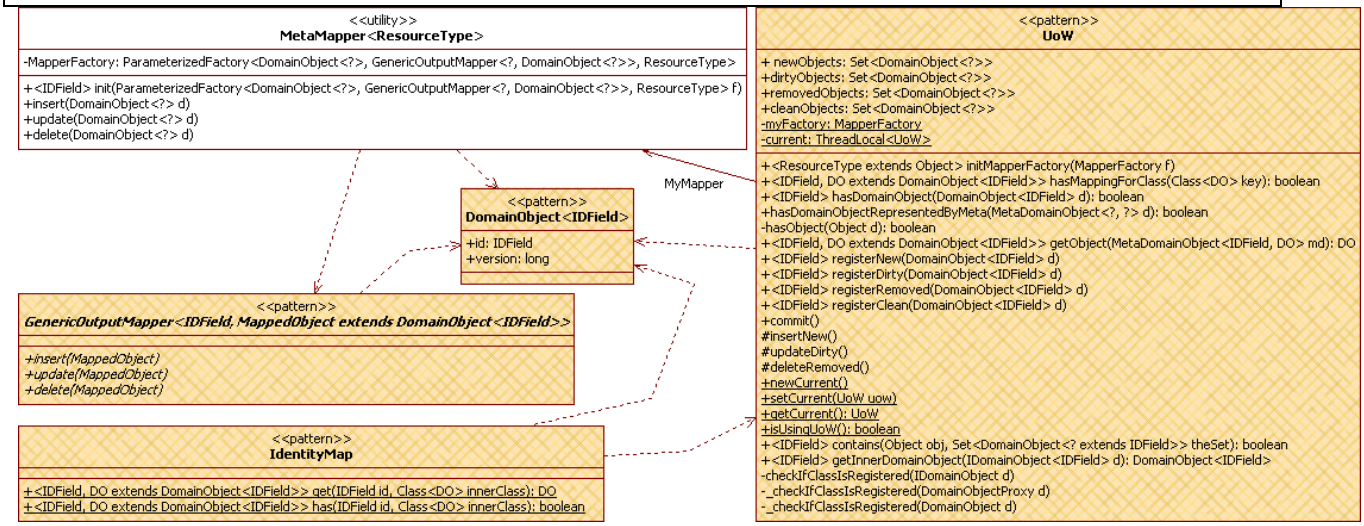


Figure 5-8 UoW backs IdentityMap

Registering Domain Objects with a UoW from within an OutputMapper will cause errors when the OutputMapper has already been called from within a UoW during its commit. Similarly, one has to be careful about calling TDGs that might change data that would also be changed by a subsequent or continued commit in UoW. The best approach is for each OutputMapper to limit which data it persists to only the DomainObject for which it is responsible.

5.2.3 UoW and IdentityMap

Our implementation of a UoW is a THREADLOCAL SINGLETON. In general, this means at most one UoW instance will exist for each request to a web Servlet[27]. The UoW defines Sets for all in-memory DomainObjects,

[27] Provided the user cleans out the ThreadLocalTracker, something that is done automatically in our Servlet implementation's postProcessRequest() method.

```
1  public static void setupUoW() {
2    MapperFactory myDomain2MapperMapper = new MapperFactory();
3    myDomain2MapperMapper.addMapping(Group.class, GroupOutputMapper.class);
4    myDomain2MapperMapper.addMapping(User.class, UserOutputMapper.class);
5    myDomain2MapperMapper.addMapping(GroupMember.class, GroupMemberOutputMapper.class);
6    UoW.initMapperFactory(myDomain2MapperMapper);
7  }
```

Figure 5-9 Sample code initializing the UoW with DomainObjects and OutputMappers

depending on their state (clean, dirty, deleted or new). As such, it makes sense to use the UoW as the back-end for an IdentityMap; the `cleanObjects` Set serves no purpose in a commit, but is used by the IdentityMap.

The UoW can distinguish between different types of DomainObjects, eliminating the need for shared sequential IDs (i.e. the UoW could accept two Chairs with `id`s 1 and 2, as well as two Tables, also with `id`s 1 and 2). DomainObjects must still be mapped to their corresponding GenericOutputMappers using a MapperFactory (Figure 5-9). This mapping is static, so all instances of the UoW will have this information available. The particulars of the MapperFactory will be discussed in section 5.4.4.

IdentityMap methods are `has(…)` and `get(…)`, which identify whether a particular DomainObject is in the UoW and retrieve that DomainObject, respectively. The methods' first parameter is the `id` to be searched for, as is normal in an IdentityMap, the second parameter is the Class of the desired DomainObject.

UoW's most important methods are:

- `newCurrent()`
  `newCurrent()` returns a new UoW which is set as the current unit of work in the THREADLOCAL SINGLETON, setting up the UoW and flushing out any previously allocated UoWs (via previous calls to `newCurrent()`).
- `setCurrent(…)`
  `setCurrent(…)`is used internally by `newCurrent()`, but can also be used as a means to let the UoW span requests (a complex activity outside the scope of this thesis).
- `getCurrent()`
  `getCurrent()` returns the current instance from the THREADLOCAL SINGLETON.
- `registerNew`/`Dirty`/`Removed`/`Clean(…)`
  The register methods register DomainObjects in the appropriate internal registries.
- `commit()`
  Calling `commit()` attempts to process new/dirty/removed DomainObjects (in that order) by calling their respective GenericOutputMappers, calling `commit()` on the data source on success and calling `rollback()`if an exception is thrown from one of the GenericOutputMapper methods.
- `initMapper(…)`
  This method statically sets up the UoW to be able to identify which GenericOutputMapper to use for each type of DomainObject. If a developer attempts to register an unmapped type, an exception indicating such is thrown.

How a developer would use UoW and IdentityMap

The most often overlooked aspect of using a UoW is its preparation[28]. In Servlets, the `init()` method of the HttpServlet (DispatcherServlet) is a convenient place to call a service method (like the one in Figure 5-9) to set up the mappings between DomainObjects and GenericOutputMappers. It is also worth noting that the compiler has sufficient information to statically check the correspondence between DomainObject and GenericOutputMapper, helping avoid runtime exceptions due to mismatches.

[28] This is based on the most frequent problems students have reported when using Unit of Work.

```
01 public static Group find(Long id)
02  throws SQLException,  DomainObjectCreationException {
03   try {
04    return IdentityMap.get(id, Group.class);
05   } catch (DomainObjectNotFoundException e) {
06   } catch (ObjectRemovedException e) {
07
08   }
09   return  getGroup(GroupTDG.find(id));
10 }
11
12 private static Group getGroup(ResultSet rs)
13  throws SQLException, MapperException, DomainObjectCreationException {
14   GroupProxy g = new GroupProxy(rs.getLong("g.id"));
15   Group result = GroupFactory.createClean(
16    rs.getLong("g.id"),
17    rs.getLong("g.version"),
18    rs.getString("g.name"),
19    new MembershipListProxy(g)
20   );
21   return result;
22 }
```

Figure 5-10 Code demonstrating the use of the UoW and IdentityMap in an InputMapper

Some setup is also required for every request, as a UoW instance must be explicitly created with the static call to `UoW.newCurrent()`. The preferred means to do this is by placing the `newCurrent()` call in the FRONT CONTROLLER implementation, for example, in Servlet class' `preProcessRequest()` method.

Once the setup of UoW is complete, there are two ways to use it. The first involves InputMapper find methods. Having already written a DomainObject and its GenericOutputMapper, a developer would then implement an InputMapper. When writing their InputMapper's `find()` method, they would check the IdentityMap for an existing DomainObject (Figure 5-10, line 4). In the event that one is not found, an instance of the DomainObject would be created after getting a ResultSet from the TDG (line 9), which would indirectly register that DomainObject as clean in the UoW by calling the `createClean()` method (line 15), as was already described in Section 5.2.1's description of DomainObjects in Figure 5-3.

The second way UoW is used is when writing DomainCommands. A developer will make use of the UoW's other register methods, registering new DomainObjects via a Factory, or explicitly registering them being deleted or updated as appropriate. At the end of such COMMANDS or possibly near the end of a Dispatcher, `UoW.commit()` will get called. Care should be taken not to write the code so that multiple commits could occur within the same request, though an explanation of why not is outside the scope of this thesis.

When `commit()` is called, each Set in the UoW is iterated through, calling the appropriate GenericOutputMapper methods on each DomainObject in that Set. It is important to be aware that the order of the Sets should be assumed to be non-deterministic; in some database systems this can cause trouble if also using FOREIGN KEY constraints. If there are MapperExceptions, `commit()` throws them back up after initiating the rollback, and the DomainCommand is responsible for any additional changes or for continuing to pass up the exception so that the Dispatcher may forward to an appropriate View to deal with the conflict.

5.2.4 ListProxy

Frequently there are one-to-many relationships between DOMAIN OBJECTS. In SoenEA these can be represented by ListProxy[29] and SetProxy. There are also several MapProxys[30] that can support additional relationships.

[29] The original List Proxy source was written by Dave Reisch, based on instructions from this thesis.
[30] The various Map Proxies were contributed by Steve Morse.

Figure 5-11 SetProxy and ListProxy

```
01  public class MembershipListProxy extends ListProxy<IGroupMembership> {
02    private IGroup myGroup;
03    public MembershipListProxy(IGroup myGroup) {
04      super();
05      this.myGroup = myGroup;
06    }
07    @Override
08    protected List<IGroupMembership> getActualList() throws Exception {
09      return GroupMembershipInputMapper.find (myGroup);
10    }
11  }
```

Figure 5-12 MembershipListProxy

In Figure 5-11 we see that ListProxy and SetProxy implement Java's List and Set interfaces. All methods from these interfaces are implemented, delegating to their `innerSet` and `innerList` respectively, via the `getInner` methods. The abstract `getActual` methods are used in the `getInner` methods to get the actual Collections.

How a developer would use a ListProxy

Figure 5-10 shows GroupInputMapper passing a new MembershipListProxy to the GroupFactory's `createNew` method. Figure 5-3 shows that the parameter is of type List<IGroupMembership>. All the developer must do is subclass ListProxy as in Figure 5-12, overriding `getActualList` and implement an appropriate constructor. It is important that all ListProxys have a field for the DomainObject that contains the List, as that will be used in retrieving the actual List from an InputMapper (line 9).

We often see that aside from returning a single DomainObject based on an ID, InputMappers are tasked with returning collections of DomainObjects. The initial set of methods of this nature (that return collections of DomainObjects), and in fact all of the InputMapper methods, can be identified simply by looking at the Proxys, be they DomainObjectProxys or one of these CollectionProxys. Developers can always follow the same implementation approach.

```
1  public static List<IGroupMembership> buildCollection(ResultSet rs, String idString)
2  throws SQLException, MapperException, DomainObjectCreationException {
3   List<IGroupMembership> l = new ArrayList<IGroupMembership>();
4   while(rs.next()) {
5    l.add(new GroupMembershipProxy(rs.getLong(idString)));
6   }
7   return l;
8  }
9
10 public static List<IGroupMembership> find(IGroup myGroup) throws SQLException,
11  MapperException, DomainObjectCreationException {
12  ResultSet rs = GroupMembershipFinder.findByGroup(myGroup.getId());
13  return buildCollection(rs, "gm.id");
14 }
```

Figure 5-13 sample code from GroupMembershipInputMapper

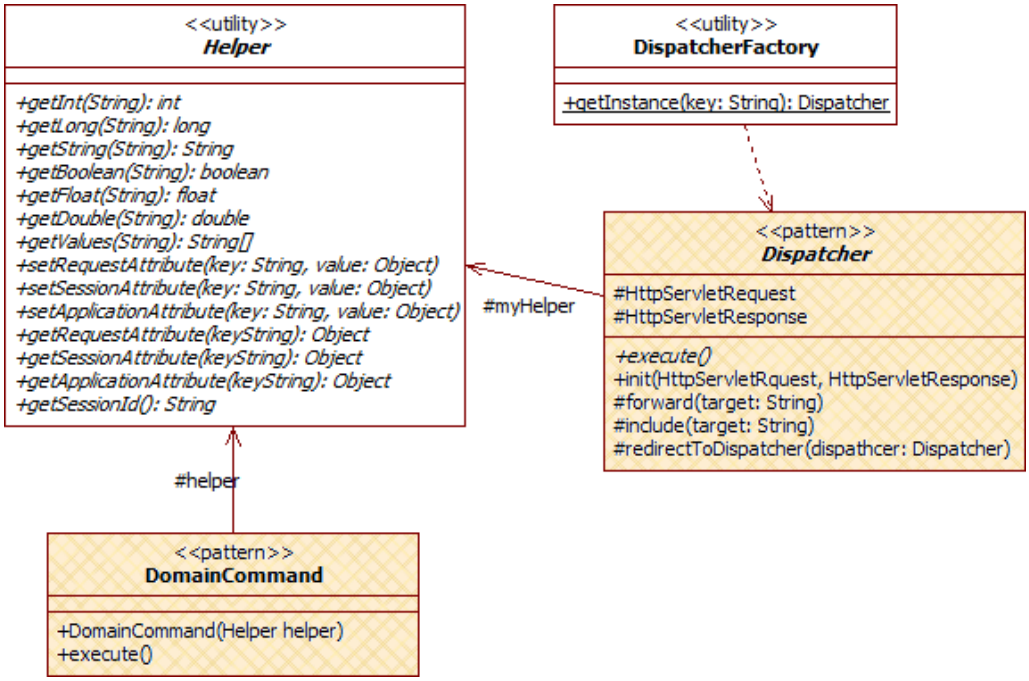


Figure 5-14 Dispatcher and DomainCommand with support classes

5.2.5 Dispatcher and DomainCommand

In making use of the DISPATCHER pattern, a user would subclass the Dispatcher class as needed. This makes available `myHelper`[31], which wraps the HttpServletRequest by default and provides the following methods:

- `init(…)`
  This method is used to prepare a Dispatcher for use. It is convenient to instantiate a Dispatcher with a default constructor when using reflection. Overriding the `init(…)` method can be done where needed, whereas a constructor must always be overridden.
- `forward(…)`
  This forwards the request to a new target, usually a JSP, but one can also forward to static content or a Servlet. The response is then generated from that target.
- `include(…)`
  As with `forward(…)`, `include(…)` can take a target. The difference is that one may only forward to content

[31] We will discuss Helpers more in Section 5.3.4

once, and upon returning from the `forward(…)` call, subsequent `forward`s/`include`s to content are forbidden. An include may be called several times on different targets.

- `redirectToDispatcher(…)`
  This is a convenience method to allow the quick chaining of Dispatchers. Passing a newly instantiated Dispatcher as a parameter to this method will automatically call its `init(…)` and then `execute()` methods.
- `execute()`
  This abstract method is a placeholder to guide the sub-classing of Dispatcher, and facilitates the dynamic dispatching mechanism that keeps FRONT CONTROLLERs oblivious to the actual Dispatchers that are called.

When using SoenEA, the general approach towards Dispatchers is to let a FRONT CONTROLLER examine the request parameters to find the canonical class name of the Dispatcher to be used for that request. DispatcherFactory's `getInstance(…)` method can then be called to dynamically create the Dispatcher. The FRONT CONTROLLER would then call the Dispatcher's `init(…)` method, passing the HttpServletRequest and HttpServletResponse. Lastly, the `execute()` method is called.

DomainCommands are similarly sub-classed by developers. The default DomainCommand implementation provides a constructor, an abstract `execute()` method, and access to the helper, mostly existing to identify a level of granularity for the developer, and to give guidance on where to start implementation.

How a developer would use Dispatcher and DomainCommand

Each activity that could be undertaken by the system user (e.g. logging in, logging out, requesting to join a group) can be implemented as a subclass of Dispatcher. Thus, one would create Dispatchers for all major use cases. The implementation of such DISPATCHERs is usually brief, calling a few DomainCommands and then deciding which JSP, File or other Dispatcher to forward to.

DomainCommands would take care of interaction with DomainObjects in order to accomplish any subtasks for the Dispatchers.

5.2.6 InputMapper

INPUT MAPPERs, as described in Section 4.5, facilitate the reading in of data from a data source and its placement into a DomainObject, much in the way an OUTPUT MAPPER does the opposite when making changes or additions to a data source.

In SoenEA there is no base implementation of an INPUT MAPPER, but their creation is part of the process for each DOMAIN OBJECT and several related patterns so they warrant mention. And while the developer is responsible for implementing the entire INPUT MAPPER, there are significant similarities in all implementations, enough to give very specific guidance to their implementation.

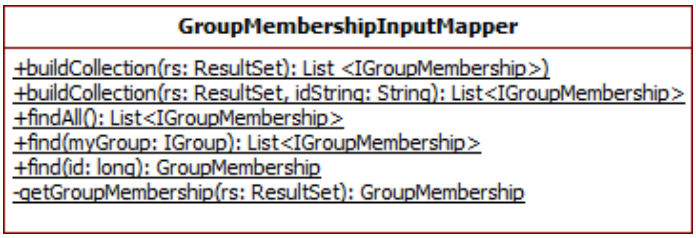


Figure 5-15 Detailed Class diagram of GroupMembershipInputMapper

Input Mappers primarily offer a variety of find methods, either geared to finding a single specific DomainObject, or a range of them. Because DomainObjects must have identity, there is always a `find()` method with a parameter for accepting the IDENTITY FIELD of the corresponding DomainObject.

Other common features in Input Mappers are methods that accept a ResultSet and transform a row of data into a specific DomainObject (usually one per InputMapper) and collection building methods that accept a ResultSet and return Collections of IDomainObjects. All these methods will be static, and with the exception of the methods that accept a ResultSet for internal use (which should be private), these methods will be public.

How a developer would implement the InputMapper Pattern

Determining what other find methods are needed is guided by their use in Commands or the various List Proxys (or other Collection Proxys) described in Section 5.2.4. These methods will take corresponding domain-specific parameters, e.g. if a Command needed to find all GroupMemberships for a Group, the GroupMembership-InputMapper would have a `find(IGroup group)` method that returns List<IGroupMembership>.

Following the approach of initially creating at most one DomainObject per find request, the method responsible for generating that DomainObject from a ResultSet should create Proxys for any other DomainObject used (Figure 5-16, lines 10 and 11). Recall that the `createClean` method called from the DomainObject's Factory will register this object as clean with the UoW.

Again, considering initially only creating at most one DomainObject per find request, the build methods will instantiate an appropriate type of Collection and then iterate over the provided ResultSet, inserting Proxys into the Collection (Figure 5-13).

```
01  private static GroupMembership getGroupMembership(ResultSet rs)
02   throws SQLException, MapperException,
03   DomainObjectCreationException {
04   Calendar cal = Calendar.getInstance();
05   cal.setTimeInMillis(rs.getLong("gm.lastUpdated"));
06   GroupMembership result =
07    GroupMembershipFactory.createClean(
08    rs.getLong("gm.id"),
09    rs.getLong("gm.version"),
10    new UserProxy(rs.getLong("gm.member")),
11    new GroupProxy(rs.getLong("gm.group")),
12    MembershipStatus.values()[rs.getInt("gm.status")],
13    cal
14   );
15   return result;
16  }
```

Figure 5-16 the getGroupMembership method

5.2.7 TDG/Finder

In Section 4.5, we highlighted our contributions of including elements of optimistic concurrency management, data sanitization and the use of a Finder on top of the TDG, elements which will be illustrated here. As in Section 5.2.6 on InputMappers, there is no generic implementation of the TDG or FINDER patterns in SoenEA, but they are part of the standard patterns used for each DOMAIN OBJECT and have a well established procedure for their implementation.

TDGs generally have 6 methods:

- insert
- update

- delete
- createTable
- dropTable
- getMaxId

The `insert` and `update` take primitive types based on the Domain Object. `delete` only takes an `id` and `version`. The `createTable` and `dropTable` methods take no parameters and allow the programmatic setting up and tearing down of database tables. The `getMaxId` method always returns an unused `id` from the system.

Finders generally have several find methods that take appropriate primitive type data, e.g. `findAll()` or `findByGroup(Long group)`, where the `group` parameter would be the `id` of the Group being looked for. Each of these find methods returns a ResultSet that can be used to get data for one or potentially many Domain Objects.

Both Finders and TDGs store their SQL Strings in static final fields. As a minor convention, we store the table name both with and without any table prefix as public static fields in the TDG, facilitating working with some other SoenEA utility classes and allowing other TDGs to build aggregate names based on the names of tables.

How a developer would implement the TDG and Finder Patterns

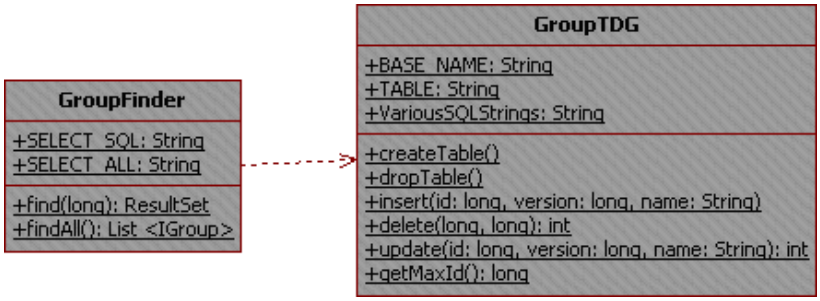


Figure 5-17 Class diagram of Group TDG/Finder

The arguments of the `insert` and `update` method are dictated entirely by the DomainObject's fields. All primitive data types in the DomainObject correspond to primitive data types in the data source. Enumerations correspond to an integer data type, and store the ordinal of the enumeration. When other DomainObjects are fields, then a data type corresponding to the IDENTITY FIELD is used (i.e. `BIGINT` in MySQL for a Long IDENTITY FIELD).

```
01 public final static String UPDATE_SQL =
02  "UPDATE " + TABLE + " " +
03  "SET version=version+1,name=? WHERE id=? and version=?;";
04
05 public static int update(long id, long version, String name)
06  throws SQLException {
07  Connection con = DbRegistry.getDbConnection();
08  PreparedStatement ps = con.prepareStatement(UPDATE_SQL);
09  ps.setString(1, name);
10  ps.setLong(2,id);
11  ps.setLong(3,version);
12  int count = SQLLogger.processUpdate(ps);
13  ps.close();
14  return count;
15 }
```

Figure 5-18 GroupTDG's update method

As discussed in Section 4.6, sanitizing data is a part of the process. Using Java's `PreparedStatement` mechanism, question marks are placeholders in the SQL `String` (Figure 5-18, line 03). Parameters are assigned using type-specific method calls, safely filling out the placeholders (Figure 5-18, lines 09-11[32]). The `PreparedStatement` mechanism prevents inappropriate data from changing the nature of the statement. If the call to `setString` includes end quotes, one can be assured that it cannot sneak in SQL code.

The execution of update statements (updates and deletes) return the number of rows updated. In Section 4.6 we identified that optimistic concurrency management should belong in TDGs, and returning the updated rows is the means by which this is done.

Often special case changes to the database are required, and they also belong in the TDG. The `createTable`/`dropTable` methods shown in Figure 5-17 are examples of this.

[32] The first argument represents the 1-based position of the placeholder in the string; the second is the data to be placed.

```
01 public static String SELECT_BY_ID_SQL =
02  "SELECT g.id,g.version,g.name FROM " +
03  GroupTDG.TABLE + " AS g "+
04  "WHERE g.id=?;";
05
06 public static ResultSet find(long id) throws SQLException{
07  Connection con = DbRegistry.getDbConnection();
08  PreparedStatement ps = con.prepareStatement(SELECT_BY_ID_SQL);
09  ps.setLong(1, id);
10  return SQLLogger.processQuery(ps);
11 }
12
13 public static String SELECT_ALL_SQL =
14  "SELECT g.id FROM " + GroupTDG.TABLE + " AS g;";
15
16 public static ResultSet findAll() throws SQLException{
17  Connection con = DbRegistry.getDbConnection();
18  PreparedStatement ps = con.prepareStatement(SELECT_ALL_SQL);
19  return SQLLogger.processQuery(ps);
20 }
```

Figure 5-19 GroupFinder's find identity find method

Finders have similar pairs of SQL Strings and methods as TDGs. A minimal implementation of a Finder will have one of these pairs for the IDENTITY FIELD of their DOMAIN OBJECT (e.g. the pair shown in Figure 5-19, lines 01 through 11). When the data source supports it, we use table aliases ("AS g" line 03 of Figure 5-19) to reduce the potential for confusion when moving around SQL code and to better identify which fields come from which tables[33].

When choosing which fields to provide in select statements, list all fields when looking up a single entry (Figure 5-19 line 2). When looking up many entries, only include the IDENTITY FIELD, as PROXYS will be generated from the returned data, an approach that may be altered in later phases, once analysis of the data usage indicates where best to optimize.

Some tables represent the one-many or many-many relationships between DOMAIN OBJECTS, explicitly represented in fields of those DOMAIN OBJECTS or not. These tables may not have INPUT or OUTPUT MAPPERS, but should have TDGs and FINDERS implemented for them as needed because the INPUT/OUTPUT MAPPERS of associated DOMAIN OBJECTS will use them.

How a developer would use the TDG and Finder Patterns

The Output Mapper's `update`/`insert`/`delete` methods accept their appropriate DOMAIN OBJECTS as parameters, and then call the corresponding TDG methods passing the values of that DOMAIN OBJECT's fields as parameters. `update`/`insert`/`delete` methods in an Output Mapper may also call other TDG methods as needed, for example TDGs that cover logging, statistics, or some forms of cascading changes in data.

Input Mappers and Factorys also make use of these classes, the former getting ResultSets, which all find methods return, and the latter calling the appropriate TDG's `getMaxId()` when a new `id` is needed during a `createNew(…)` call.

TDGs and Finders will also often get used by setup scripts, and even by Commands and service threads to effect database changes that are generally less explicitly tied to the concept of a DOMAIN OBJECT, or which affect one or more DOMAIN OBJECTS indirectly (e.g. updating a database-cached value of the number of times a Message DOMAIN OBJECT has been viewed, or expiring time sensitive entries).

[33] Our experience has indicated that this clarification avoids more errors than are created by copy/pasting SQL strings and having to change the table aliases in the Finders and InputMappers.

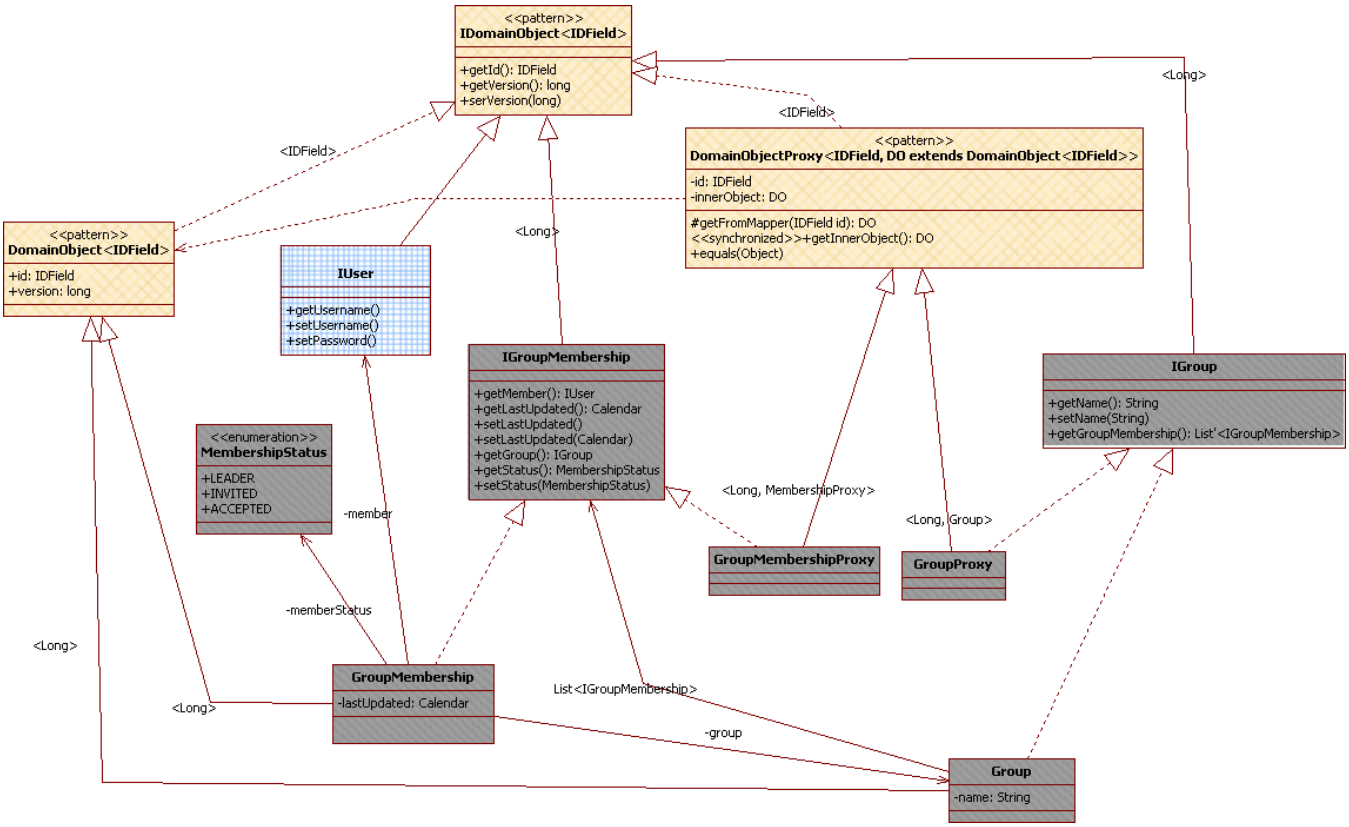


Figure 5-20 Summary of Domain Objects seen in this section

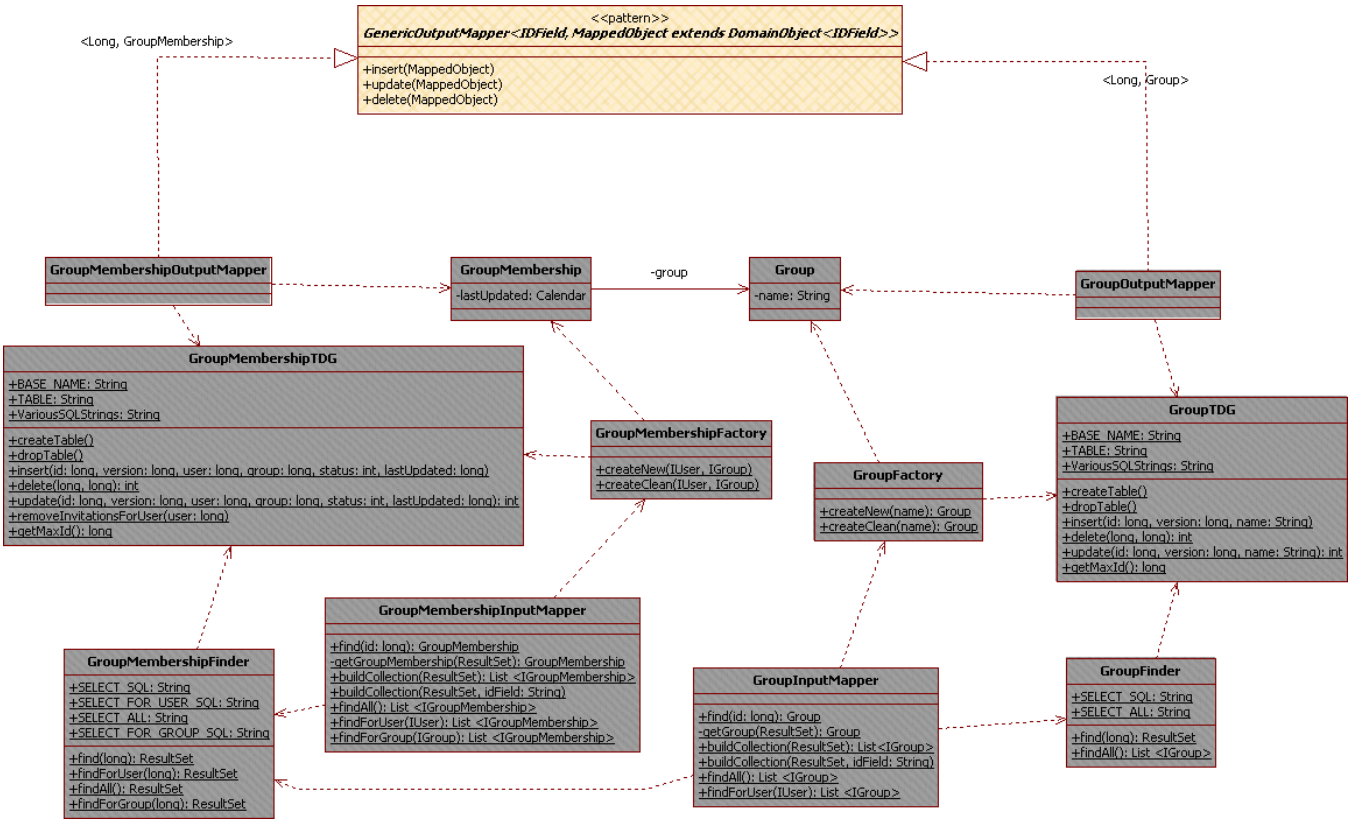


Figure 5-21 A simple class diagram showing the domain layer relating to the technical services layer

5.3 SoeanEA DITCs

We have noted that we repeat some components frequently in the development of web applications. They are generally implementations of patterns discussed in this paper, and appear prominently in their applications. Eventually we began to create base implementations that could be used as-is or sub-classed which saves significant development time. This section quickly runs through some of these and how they are expected to be used.

5.3.1 User

User is a common term in web applications. This DomainObject stores username and password information in a database and keeps track of a User's Roles. SoenEA comes with the DomainObject, its interface, Proxy, Factory, Mappers, TDG and Finder. Common subclasses strengthen how passwords are stored, but most systems can make use of it as is.

The current User is usually stored in a session attribute, thus allowing it to be checked throughout the user's session.

5.3.2 Role

A common approach in web applications is to only allow certain Users to do certain things. The use of the Role DomainObject in SoenEA is used to represent permanent and high-level groupings of these behaviors, e.g. Guest, Admin or Registered.

Role is not treated the same way as most DomainObjects in SoenEA's implementation, and demonstrates that instead of storing a DomainObject in a database, one can keep more permanent data elsewhere. As such, there is neither a TDG or Finder, but there is an altered Factory (which is more traditional as it does not use UoW and hands out SINGLETONS), a default GuestRole implementation, and the ApplicationAuthorization class to allow the quick checking of application-level authorization for a User.

MyResources.propeties

```
ConcreteRole_1=org.dsrg.soenea.domain.role.impl.GuestRole
```

Access.xml

```
<role name="org.dsrg.soenea.domain.role.impl.GuestRole">
 <command name="application.dispatcher.group.ViewGroup" get="true" />
</role>
```

Figure 5-22 MyResources.properties and Access.xml

Concrete Roles need to be registered in the MyResources.properties file (Figure 5-22), as does the location of an access XML file[34]. Once that is done, ApplicationAuthorization can be used to restrict which Dispatchers are called based on the User currently active in a request's session. ApplicationAuthorization expects a simple XML format identifying each potential Role as well as identifying which Dispatchers[35] are allowed for each Role, and which HTTP methods are accepted for each Dispatcher. If it finds a match, `ApplicationAuthorization.hasAuthority(…)` returns true, otherwise it returns false.

5.3.3 DispatcherServlet and Servlet

Servlet acts as our default implementation of a CONTROLLER. It organizes the flow of the call from Tomcat, hiding the difference between `get` calls and `post` calls, sets some initial values for a recommended error VIEW, and calls setup and cleanup items to ensure the smooth running of a system, ensuring database connections are closed and ThreadLocal items are appropriately cleaned out between requests.

Normally, a developer would still need to write code to implement the part of their CONTROLLER that called DISPATCHERS, so DispatcherServlet was created to allow a default implementation of a FRONT CONTROLLER without any authorization. Any application that is simple and does not require more than one database connection could use DispatcherServlet, as is, for a FRONT CONTROLLER.

5.3.4 Helper

SoenEA's HELPER facilitates use of the HttpServletRequest. It wraps it and provides convenience methods to access attributes in various contexts, it provides type specific access to passed parameters (`getBoolean`/`getLong`/etc.), and it provides easy access to a session ID. SoenEA's specific implementation is HttpServletHelper.

The default implementation is sufficient in most cases, however where multi-part posts are made, one needs to implement a custom HELPER.

5.4 SoenEA Utilities

To support the pattern classes, as well as the DITCs, and just for general use, SoenEA has several utility classes. We will quickly touch on the purpose of the most frequently seen utilities, as they are primarily of practical interest in this thesis. Any theory behind them is outside the scope of this thesis.

[34] This is relative to the context of the application, usually `WEB-INF/classes/Access.xml`
[35] Or FrontCommands, which is why `command` is used instead of `dispatcher`, as our SOEN students had learned this first.

5.4.1 DispatcherFactory

The DispatcherFactory creates instances of the Dispatcher class when passed a canonical class name. We generally pass the full canonical class name of Dispatchers to Front Controllers instead of making up an artificial key.

5.4.2 UniqueIdFactory, UniqueIdTDG/Finder

The IDENTITY FIELD is a required aspect of DOMAIN OBJECTS, as discussed in Sections 4.1 and 5.2.1. DOMAIN OBJECT Factorys will need IDs when creating new Domain Objects, and they will accordingly call the appropriate DOMAIN OBJECT's TDG method, `getMaxId()`, as new IDs are needed. This `getMaxId()` method should make use of an appropriate `UniqueIdFactory` to accomplish this task (Figure 5-23).

We have created two types of Unique ID FACTORY: one is suited to applications that are the sole means of accessing a database; the other can be used with any number of applications accessing the same data while still providing each with Unique IDs. We call these two FACTORYS SingleAppUniqueIdFactory and MultiAppUniqueIdFactory respectively. By default, UniqueIdFactory will return a SingleAppUniqueIdFactory, but the Factory that it uses internally can be set as desired.

```
public static long getMaxId() throws SQLException {
 return UniqueIdFactory.getMaxId( BASE_NAME, "id" );
}
```

Figure 5-23 Use of a UniqueIdFactory

MultiAppUniqueIdFactory ensures unique IDs across applications by locking an ID table. This means the overhead of using MultiAppUniqueIdFactory is an additional database connection. The access to this MultiAppUniqueIdFactory's ID table is done through the UniqueIdTDG and UniqueIdFinder. SingleAppUniqueIdFactory requires no additional table or connection.

5.4.3 MetaDomainObject

The motivation for the use of the MetaDomainObject is to be able to check for equality. By creating a MetaDomainObject with ID and class fields, a developer can then test for equality with an actual DomainObject.

MetaDomainObject is primarily used as a convenience class by IdentityMap and UoW. It is meant to simplify the code used for looking up DomainObjects in the UoW.

5.4.4 MapperFactory and MetaMapper

These utilities allow the UoW to call the correct OUTPUT MAPPER implementation for a given DomainObject that is registered with it (e.g. `insertNew()` calling `insert` methods for each DomainObject that was registered new in Figure 5-24, lines 10-15). It does so by storing a MapperFactory, which acts as a Map of GenericOutputMappers, which all DomainObject OUTPUT MAPPERS implement. This map is populated by MapperFactory's `addMapping` method, usually in a Front Controller's initialization (as previously shown in Figure 5-9).

```
01 private static MapperFactory myFactory;
02
03 public static <ResourceType extends Object> void
04  initMapperFactory(MapperFactory f) {
05  MyMapper = new MetaMapper<ResourceType>();
06  MyMapper.init(f);
07  myFactory = f;
08 }
09
10 protected void insertNew() throws SQLException,
11  KeyNotFoundException, CreationException, MapperException {
12  for (DomainObject<?> d: newObjects) {
13   MyMapper.insert(d);
14  }
15 }
```

Figure 5-24 UoW using MapperFactory and MetaMapper

MetaMapper provides a simple interface over this process, allowing UoW's use of the MapperFactory to be more intuitive.

5.4.5 DBRegistry, ConnectionFactorys and Connections

Most frameworks have some means of interacting with databases easily. SoenEA makes use of a ThreadLocal registry, DBRegistry. DBRegistry keeps track of ConnectionFactorys that are used to identify how a database connection is made. For a given Thread, DBRegistry will keep track of any Connections created through DBRegistry for the duration of the Thread.

Two database-specific Connections have been implemented, one for MySQL and one for Derby. Their purpose is to wrap some of the database specific differences in such a way that developers can overlook them. Currently, only write locks on tables are supported in this fashion. Otherwise, SoenEA Connections behave exactly like java.sql.Connection.

5.4.6 ApplicationAuthorization

ApplicationAuthorization is a utility that allows quick determination of whether a List of Roles has access to a given command for a given HTTP method. Generally, the command is a canonical class name of a Dispatcher, but it could in theory represent any String required for authentication.

ApplicationAuthorization offers a means to apply "Application Level Authentication", a term we use to represent the restriction of access to a behavior at the application level, vs. in the GUI, where one would be effectively restricting visibility, or in the Domain layer where one would check for access in Commands. While this concept is one that we explore in many of the implementations using SoenEA, it is outside the scope of this thesis.

5.4.7 ThreadLocalTracker

The re-use of Threads to serve multiple requests is extremely common. In Tomcat, for example, one cannot assume that a subsequent request to the server will spawn a new Thread, as it may just re-use an existing thread that is not currently serving another request.

The concept of ThreadLocal is otherwise very useful to act as a point of access to data that needs to be used throughout an application, and which would otherwise clutter method calls with extra parameters. To resolve the problem, we register ThreadLocal instances used by requests with the ThreadLocalTracker, and at the end of each request, we ensure that they are purged. This is similar to how we ensure that our database connections are closed, so that the next Thread may start fresh.

5.4.8 Exceptions

SoenEA comes with a few general use Exceptions to better classify common problems that may happen. Here we will outline some of the more commonly used ones:

- MapperException
- DomainObjectCreationException
  This exception is thrown when an expected creation of a DomainObject fails. This Exception could be thrown when a request erroneously asks for an `id` that is not in the database. This is often a sign that DomainObjects are not being cleaned up properly after `delete`s or `update`s, or that the wrong set of `id`s are being used to pull up DomainObjects.
- DomainObjectNotFoundException
  This is an Exception thrown by IdentityMap when a call to get a DomainObject is made and the `id` and class are not in the map. This Exception should be thrown whenever a call on IdentityMap for the same `id` and class would return false.
- LostUpdateException
  This is an Exception that should be thrown by GenericOutputMappers when they identify that a lost update has occurred. Usually, an OUTPUT MAPPER would check the result of a `delete` or `update` to ensure that the resulting changed rows are not 0, but could also be caused by other more complex problems (as certain race conditions in poorly ordered database requests can lead to MySQL throwing a deadlock exception which represents a failed lost update).
- CommandException
  Most problems that occur in a Command should throw a CommandException. Based on the CommandException thrown, the Dispatcher should be able to determine the appropriate additional Commands or Views to dispatch to.
- ProxyException
  A ProxyException is thrown when one of the CollectionProxys is unable to instantiate its `innerList`. This is generally caused by some form of database problem in the InputMapper `find` method being called to build the `innerList`.

5.5 SoenEA Test Suite

SoenEA comes with an ever growing test suite. There are currently 11 regular test classes and 6 regression test classes, covering 50 different unit tests. To support these tests, several concrete implementations of the patterns are included. While these are distributed with SoenEA for the purpose of testing, they are not considered DITCs, although they still offer good examples for simple implementation.

6 Professional Software Development Using SoenEA

Throughout the writing of this thesis, we have been applying the theory described herein and implementing using SoenEA on a variety of projects. Both the theory and SoenEA have been constantly changing based on our experience, but we have confirmed the commercial viability of both. Here are a few:

- J-Site
  This is a complex forum/internal messaging/gallery/CMS system with the potential to expand to support other features or systems as needed. It went into production for several months, at its peak supporting between 1 and 2 million hits/day and having 6000 distinct registered users logging in per day and even more guest users. It was developed by Stuart Thiel with some development done by contract employees. (46k SLOC, 10 months development)
- Cubique
  This is a portal system with file management/internal messaging/user management, and is still in development. It is being developed by Concordia's Bioinformatics Lab and is being overseen by Stuart Thiel. (37k SLOC, 13 months development)
- YP Listings
  A yellow pages listing service that has been in production for two years and supports several million records. It was developed by Stuart Thiel with some development done by contract employees. (24k SLOC, 8 months development)
- Korsakow5
  This is an open source desktop application for the creation of non-linear video narrative presentations. It was developed by David Reisch, designed and managed until initial release by Stuart Thiel, and funded by CINER-G (Concordia Interactive Narrative Experimentation and Research Group). The original concept and development was by Florian Thalhofer. (59k SLOC, 14 months development — 8 until initial release)

7 Conclusion

Our approach has been very successful in practice. We have trained developers to reliably create effective software that meets requirements, has few bugs, and is easier to test and to review than most other WEA code that we have explored. Primarily, the developers introduced to SoenEA and the approach presented in this thesis have been for novices and students. As such, we have focused on making the system easy to learn and use, while not oversimplifying to the point of being useless. Given both the success in training new developers with this software and the viability of applications that have been developed using the framework and prescribed approach, we are confident that there is a future for this work.

The goal of refining the existing toolkit of patterns, has been hugely successful. While our experience is biased towards new developments, the changes we have introduced are founded in simple and reliable development principles and the result is accordingly simple and reliable.

The biggest achievement, and a major reason for our success in training developers, is the more concrete guidance in approach. A developer who has a UML Domain Model worked out along with some Use Cases can almost deterministically generate all the code they require without having to make any complex decisions until a working system is running under enough load to provide performance data. Accompanying this practical guidance is SoenEA, which further reinforces the recommended approach while limiting tedious implementation tasks. This combination has greatly exceeded our expectations.

Future Work

There are several important areas where the theory presented in this thesis could be expanded. We have already begun work on how testing fits in with this approach, in particular we have begun work on a system-level testing approach that is tied closely to both DISPATCHERs and Use Cases.

Once a systematic testing approach exists, the next areas to explore are the advanced topics needed to progress a project to commercial readiness. This would cover the theory of analyzing the performance of a system, given the domain-oriented approach we promote, and the subsequent optimization of the access to data. To complement that theory, some guidance on the specifics of working with SQL and the general types of SQL statements/optimization/problems frequently come across when working with Domain Objects world be instructive.

Marek Krajewski, working with the Concordia Bioinformatics lab, has been working on a project titled DOCrib to help codify the relationship between DOMAIN OBJECTs, INPUT MAPPERs and TDGs. DOCrib has the potential to further improve on the implementation approach suggested by SoenEA by reducing much of the duplication in TDGs and FINDERs and making the querying of ResultSets in INPUT MAPPERs more intuitive.

The advent of AJAX has given WEAs the responsiveness of desktop applications, and the flexibility for incredibly rich interfaces, but there is little guidance available on the integration of such GUIs with a back-end system. An examination of the existing Javascript frameworks/toolkits available today would further help WEA developers, as would a comprehensive examination of UI design patterns.

As much of the theory described in this thesis leads to a nearly deterministic approach to implementation, there has already been some work in terms of code generation. Asif Dogar's Master's thesis, *Model Driven Development for Enterprise Applications* [DOGAR 2007], demonstrated a simplistic approach to code generation based on SoenEA using Rational Software Architect. More recently, Brendan Asselstine, working for the Concordia Bioinformatics lab, developed SeaDog: a more sophisticated code generation tool with a web interface that allows the visual creation of Domain Models in UML and their subsequent transformation into an implementation that uses SoenEA and follows the suggested approaches from the thesis. SeaDog is incomplete, but is already far enough along that some developers have successfully used it to create a basic class structure for some simple projects. As of January 2010, the Concordia Bioinformatics lab has allocated another programmer to advance the development of SeaDog.

In terms of WEA development, we have begun to note a large number of Application patterns. We have already begun to experiment with these high-level patterns, such as the Forum/Thread/Message model used in J-Site and the Auditing model used in Cubique. The newest version of SoenEA has some abstract classes for the use of Forums/Threads/Messages. However, the implications of such high level patterns are broad, and it is easy to fall into

the trap of defining a pattern that is implementation-specific. Further analysis on how to describe such patterns is necessary, if we can even continue to call it a pattern.

As with implementation using this approach and SoenEA, one always knows the next step. In good engineering, the satisfaction comes not in finding a solution, but in implementing that solution expertly. We hope to continue improving on this work in that spirit.

8 References

[Fowler 2003] Martin Fowler, *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2003, ISBN: 0321127420

[MartinFowler] "P of EAA: Domain Model" http://www.martinfowler.com/eaaCatalog/domainModel.html, Jan. 24, 2010

[EBAY] "eBay" http://www.ebay.ca/

[AMAZON] "Amazon.com" http://www.amazon.com/

[Tomcat] "Apache Tomcat" http://tomcat.apache.org/

[Hibernate] "Hibernate" http://www.hibernate.org Mar. 30, 2008

[Struts] "Apache Struts" http://struts.apache.org/ Jan. 24, 2010

[EJB] "Enterprise Java Beans" http://java.sun.com/products/ejb/ Mar. 30, 2008

[Sutherland97] "The Object Technology Architecture: Business Objects for Corporate Information System" http://jeffsutherland.com/papers/boa_pap.html May. 01, 2008

[Larman 2004] Larman, Craig [2004] (2005). *Applying UML and Patterns - An Introduction to Object-Oriented Analysis and Design and Iterative Development*, 3rd, Prentice Hall PTR. ISBN 0-13-148906-2.

[YP] "EasYellowPages.com development site" http://ns2.htmlweb.com/yp/search.html Uses SoenEA/commercial

[MovieMapper] "Brand Hype" http://www.brandhype.org Uses SoenEA/ written in collaboration with a communications prof. Uses an older version of SoenEA (was the testbed for the jump from java4 to java5).

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*.

[Alur 2001] Deepak Alur, John Crupi, Dan Malks (2001), *Core J2EE Patterns: Best Practices and Design Strategies*, Prentice Hall / Sun Microsystems Press, 2nd Edition. ISBN:0130648841

[JSP] "JavaServer Pages Technology" http://java.sun.com/products/jsp/ Feb. 09, 2009

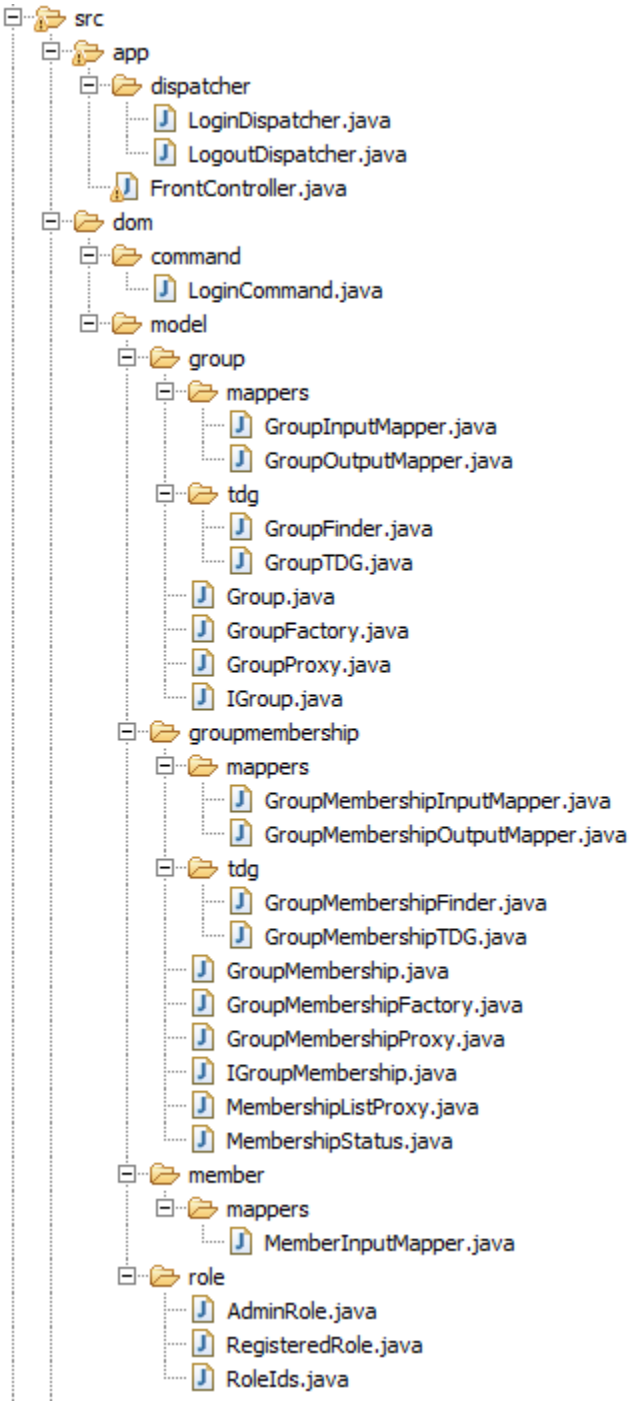
[SERVLET] "Java Servlet Technology" http://java.sun.com/products/servlet/ Feb. 09, 2009

[DOGAR 2007] Dogar, Asif *Model Driven Development for Enterprise Applications*. M.A. thesis, Concordia University Computer Science and Software Engineering Department, Montreal, Canada (2007)

[USCENSUS] "U.S. Census Bureau E-Stats" http://www2.census.gov/retail/releases/historical/ecomm/09Q2.html Jan. 23, 2010

9 Appendix

In this Section we will provide the initial files for the WEA described in Chapter 5. The file structure will be shown to provide an example of how an implementation can be structured within an eclipse project. All the Java source, as well as the Access.xml and MyResources.properties files will be included. A digital version will be made available under https://soenea.htmlweb.com/trac/browser/SoenEA/trunk/documentation/Thesis/trunk



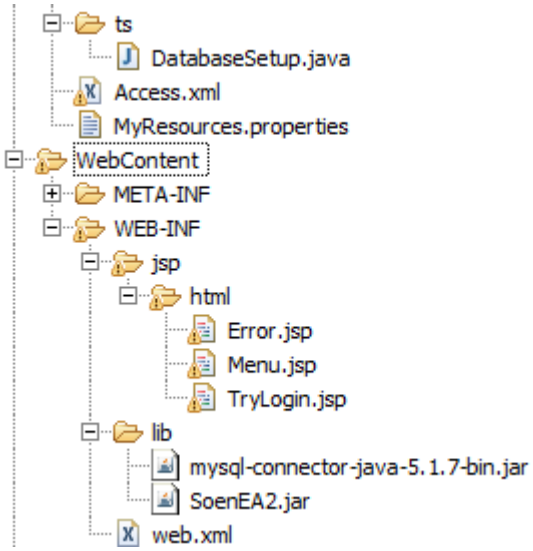


Figure 9-1 A recommended directory structure

```
001 package app;
002
003 //imports
004
005 public class FrontController extends DispatcherServlet {
006
007  private static final long serialVersionUID = 1L;
008  public static final String LOG_STRING = "soenea.test";
009  private static String defaultDispatcher = "";
010
011  @Override
012  public void init(ServletConfig config) throws ServletException {
013   super.init(config);
014   try {
015    defaultDispatcher = Registry.getProperty("defaultDispatcher");
016   } catch (Exception e1) {
017
018   }
019   ApplicationAuthorizaton.setBasePath(getServletContext()
020     .getRealPath("."));
021   prepareDbRegistry();
022
023   setupUoW();
024
025   /**
026    * My attempts to use Logging commons utilities
027    */
028   String loglevel = null;
029   try {
030    loglevel = Registry.getProperty("LogLevel");
031    if (loglevel.trim().equals(""))
032     throw new Exception("EmptyProperty");
033   } catch (Exception e1) {
034    loglevel = "INFO";
035   }
036
037   try {
038    for (Handler h : Logger.getLogger(FrontController.LOG_STRING)
039      .getHandlers()) {
040     Logger.getLogger(FrontController.LOG_STRING).removeHandler(h);
041    }
042
043    ConsoleHandler ch = new ConsoleHandler();
044    ch.setLevel(Level.ALL);
045    Logger.getLogger(FrontController.LOG_STRING).addHandler(ch);
046    Logger.getLogger(FrontController.LOG_STRING).setLevel(
047      Level.parse(loglevel));
048    Logging.setLoggerString(FrontController.LOG_STRING);
049    SQLLogger.setLegThreshold(500);
050   } catch (SecurityException e) {
051    // TODO Auto-generated catch block
052    e.printStackTrace();
053   }
054
055  }
056
057  public static void setupUoW() {
058   MapperFactory myDomain2MapperMapper = new MapperFactory();
```

```
059
060   myDomain2MapperMapper.addMapping(User.class, UserOutputMapper.class);
061   myDomain2MapperMapper.addMapping(GuestUser.class,
062     UserOutputMapper.class);
063   myDomain2MapperMapper.addMapping(GuestRole.class,
064     RoleOutputMapper.class);
065   myDomain2MapperMapper.addMapping(AdminRole.class,
066     RoleOutputMapper.class);
067   myDomain2MapperMapper.addMapping(RegisteredRole.class,
068     RoleOutputMapper.class);
069
070   UoW.initMapperFactory(myDomain2MapperMapper);
071  }
072
073  public static void prepareDbRegistry() {
074   prepareDbRegistry("");
075  }
076
077  public static void prepareDbRegistry(String key) {
078   MySQLConnectionFactory f = new MySQLConnectionFactory(null, null, null,
079     null);
080   try {
081    f.defaultInitialization();
082   } catch (SQLException e2) {
083    // TODO Auto-generated catch block
084    e2.printStackTrace();
085   }
086   DbRegistry.setConFactory(key, f);
087   String tablePrefix;
088   try {
089    tablePrefix = Registry.getProperty(key + "mySqlTablePrefix");
090   } catch (Exception e1) {
091    // TODO Auto-generated catch block
092    e1.printStackTrace();
093    tablePrefix = "";
094   }
095   if (tablePrefix == null) {
096    tablePrefix = "";
097   }
098   DbRegistry.setTablePrefix(key, tablePrefix);
099  }
100
101  @Override
102  protected void processRequest(HttpServletRequest request,
103    HttpServletResponse response) throws ServletException,
104    java.io.IOException {
105   request.setCharacterEncoding("UTF-8");
106   Helper myHelper = null;
107   Dispatcher command = null;
108   String commandName = null;
109   IUser user = null;
110   try {
111
112    for (Object key : request.getParameterMap().keySet()) {
113     Logger.getLogger(LOG_STRING).log(
114       Level.FINEST,
115       ("Key: " + key + " Value: " + Arrays.toString(request
116         .getParameterValues(key.toString()))));
```

```
117    }
118    commandName = getCommandName(request);
119
120    if (commandName == null)
121     commandName = "";
122
123    user = (IUser) myHelper.getSessionAttribute("CurrentUser");
124
125    if (user == null) {
126     user = new GuestUser();
127     request.getSession(true).setAttribute("CurrentUser", user);
128    }
129
130    if (!(user instanceof GuestUser)) {
131     user = UserInputMapper.find(user.getId());
132     request.getSession(true).setAttribute("CurrentUser", user);
133    }
134
135    if (!ApplicationAuthorizaton.hasAuthority(commandName, user
136      .getRoles(),
137      ApplicationAuthorizaton.RequestMethod.valueOf(request.getMethod()))) {
138        throw new Exception("Access Denied to " + commandName
139       + " for user " + user.getUsername());
140    }
141    command = DispatcherFactory.getInstance(commandName);
142
143    Logger.getLogger(LOG_STRING).log(Level.FINE,
144      command.getClass().toString());
145    command.init(request, response);
146    long time = System.currentTimeMillis();
147    command.execute();
148    Logger.getLogger(LOG_STRING).log(
149      Level.FINER,
150      "Time to execute command: "
151        + (System.currentTimeMillis() - time) + "ms.");
152   } catch (Exception exception) {
153    Throwable e = exception;
154    Logger.getLogger(LOG_STRING).throwing(getClass().getName(),
155      "processRequest", e);
156    request.setAttribute("errorMessage", e.getMessage());
157    request.setAttribute("exception", e);
158    request.getRequestDispatcher("/WEB-INF/JSP/html/Error.jsp")
159      .forward(request, response);
160   }
161  }
162
163  @Override
164  protected void preProcessRequest(HttpServletRequest request,
165    HttpServletResponse response) {
166   super.preProcessRequest(request, response);
167   UoW.newCurrent();
168   try {
169    DbRegistry.getDbConnection().setAutoCommit(false);
170    DbRegistry.getDbConnection().createStatement().execute(
171      "START TRANSACTION;");
172   } catch (SQLException e) {
173    e.printStackTrace();
174   }
```

```
175  }
176
177  @Override
178  protected void postProcessRequest(HttpServletRequest request,
179    HttpServletResponse response) {
180   try {
181    DbRegistry.getDbConnection().createStatement().execute("ROLLBACK;");
182    DbRegistry.getDbConnection().close();
183    DbRegistry.closeDbConnectionIfNeeded();
184   } catch (Exception e) {
185    e.printStackTrace();
186   }
187   ThreadLocalTracker.purgeThreadLocal();
188  }
189
190  protected String getCommandName(HttpServletRequest request)
191    throws Exception {
192   String commandName = request.getParameter("command");
193   if (commandName == null || commandName.equals("")) {
194    if (defaultDispatcher == null)
195     throw new Exception("HTTP attribute 'command' is missing.");
196    else
197     commandName = defaultDispatcher;
198   }
199   return commandName;
200  }
201
202 }
```

Figure 9-2 FrontController

```
001 package app.dispatcher;
002
003 //imports
004
005 public class LoginDispatcher extends Dispatcher {
006
007  @Override
008  public void execute() throws ServletException, IOException {
009   try {
010    new LoginCommand(myHelper).execute();
011    forward("/WEB-INF/JSP/html/Menu.jsp");
012   } catch (Exception e) {
013    forward("/WEB-INF/JSP/html/TryLogin.jsp");
014   }
015  }
016
017 }
```

Figure 9-3 LoginDispatcher

```
001 package app.dispatcher;
002
003 //imports
004
005 public class LogoutDispatcher extends Dispatcher {
006  @Override
007  public void execute() throws ServletException, IOException {
008   try {
009    myRequest.getSession().invalidate();
010    myHelper.setSessionAttribute("CurrentUser", new GuestUser());
011    forward("/WEB-INF/JSP/html/TryLogin.jsp");
012   } catch (Exception e) {
013    forward("/WEB-INF/JSP/html/TryLogin.jsp");
014   }
015  }
016
017 }
```

Figure 9-4 LogoutDispatcher

```
001 package dom.command;
002
003 //imports
004
005 public class LoginCommand extends Command {
006
007  public LoginCommand(Helper helper) {
008   super(helper);
009  }
010
011  @Override
012  public void execute()
013    throws CommandException {
014   String username = helper.getString("username");
015   String password = helper.getString("password");
016
017   if(username == null && password == null) throw new CommandException("");
018
019   try {
020    helper.setSessionAttribute("CurrentUser",
021      UserInputMapper.find(username, password));
022
023   } catch (SQLException e) {
024
025    e.printStackTrace();
026    throw new CommandException(e);
027   } catch (MapperException e) {
028    getNotifications().add("Sorry, no user for that " +
029      "username and password combo.");
030    throw new CommandException("Sorry, no user for that " +
031      "username and password combo.");
032   }
033
034  }
035 }
```

Figure 9-5 LoginCommand

```
001 package dom.model.group;
002
003 //imports
004
005 public interface IGroup extends IDomainObject<Long> {
006
007  public abstract String getName();
008
009  public abstract void setName(String name);
010
011  public List<IGroupMembership> getMembers();
012
013  public void setMembers(List<IGroupMembership> members);
014 }
```

Figure 9-6 IGroup

```
001 package dom.model.group;
002
003 //imports
004
005 public class Group extends DomainObject<Long> implements IGroup {
006  private String name;
007  private List<IGroupMembership> members;
008
009  protected Group(Long id, long version, String name, List<IGroupMembership> members) {
010   super(id, version);
011   this.name=name;
012   this.members=members;
013  }
014
015  public String getName() {
016   return name;
017  }
018  public void setName(String name) {
019   this.name = name;
020  }
021
022  public List<IGroupMembership> getMembers() {
023   return members;
024  }
025
026  public void setMembers(List<IGroupMembership> members) {
027   this.members = members;
028  }
029 }
```

Figure 9-7 Group

```
001 package dom.model.group;
002
003 //imports
004
005 public class GroupProxy extends DomainObjectProxy<Long, Group> implements
006   IGroup {
007
008  public GroupProxy(Long id) {
009   super(id);
010  }
011
012  @Override
013  protected Group getFromMapper(Long id) throws SQLException,
014    DomainObjectCreationException {
015   try {
016    return GroupInputMapper.find(id);
017   } catch (MapperException e) {
018    throw new DomainObjectCreationException(e.getMessage(),e);
019   }
020  }
021
022  @Override
023  public String getName() {
024   return getInnerObject().getName();
025  }
026
027  @Override
028  public void setName(String name) {
029   getInnerObject().setName(name);
030  }
031
032  @Override
033  public List<IGroupMembership> getMembers() {
034   return getInnerObject().getMembers();
035  }
036
037  @Override
038  public void setMembers(List<IGroupMembership> members) {
039   getInnerObject().setMembers(members);
040  }
041
042 }
```

Figure 9-8 GroupProxy

```
001 package dom.model.group;
002
003 //imports
004
005 public class GroupFactory  {
006  public static Group createNew(String name, List<IGroupMembership> members) throws SQLExceptior
007   Group result = new Group(GroupTDG.getMaxId(), 0l, name, members);
008   UoW.getCurrent().registerNew(result);
009   return result;
010  }
011
012  public static Group createClean(long id, long version, String name, List<IGroupMembership>
members)
013   throws SQLException {
014   Group result = new Group(id, version, name, members);
015   UoW.getCurrent().registerClean(result);
016   return result;
017  }
018 }
```

Figure 9-9 GroupFactory

```
001 package dom.model.group.mappers;
002
003 //imports
004
005 public class GroupInputMapper {
006
007  public static List<IGroup> buildCollection(ResultSet rs)
008   throws SQLException, MapperException, DomainObjectCreationException {
009   return buildCollection(rs, "g.id");
010  }
011  public static List<IGroup> buildCollection(ResultSet rs, String idString)
012   throws SQLException, MapperException, DomainObjectCreationException {
013   ArrayList<IGroup> l = new ArrayList<IGroup>();
014   while(rs.next()) {
015    l.add(new GroupProxy(rs.getLong(idString)));
016   }
017   return l;
018  }
019
020  public static List<IGroup> findAll() throws SQLException, MapperException,
DomainObjectCreationException {
021   ResultSet rs = GroupFinder.findAll();
022   return buildCollection(rs);
023  }
024
025  public static Group findByName(String name) throws SQLException, MapperException,
026   DomainObjectCreationException {
027   ResultSet rs = GroupFinder.findByName(name);
028
029   if(!rs.next()) throw new MapperException("The record for this Group id doesn't exist");
030   try {
031    return IdentityMap.get(rs.getLong("g.id"), Group.class);
032   } catch (DomainObjectNotFoundException e) {
033   } catch (ObjectRemovedException e) {
034
035   }
036
037   return getGroup(rs);
038  }
039
040  public static Group find(long id) throws SQLException, MapperException,
DomainObjectCreationException {
041   try {
042    return IdentityMap.get(id, Group.class);
043   } catch (DomainObjectNotFoundException e) {
044   } catch (ObjectRemovedException e) {
045
046   }
047   ResultSet rs = GroupFinder.find(id);
048   if(!rs.next()) throw new MapperException("The record for this Group id doesn't exist");
049   return getGroup(rs);
050  }
051
052
053  private static Group getGroup(ResultSet rs) throws SQLException, MapperException,
054   DomainObjectCreationException {
055   long id = rs.getLong("g.id");
056   IGroup thisGroup = new GroupProxy(id);
057   Group result = GroupFactory.createClean(
058     id,
059     rs.getLong("g.version"),
060     rs.getString("g.name"),
061     new MembershipListProxy(thisGroup)
062    );
063   return result;
```

```
001 package dom.model.group.mappers;
002
003 //imports
004
005 public class GroupOutputMapper implements
006   GenericOutputMapper<Long, Group> {
007
008  @Override
009  public void delete(Group group) throws MapperException {
010   try {
011    int count = GroupTDG.delete(group.getId(), group.getVersion());
012    if(count == 0) throw new LostUpdateException("GroupTDG: id " + group.getId() +
013     " version " + group.getVersion());
014    group.setVersion(group.getVersion()+1);
015   } catch (SQLException e) {
016    throw new MapperException("Could not delete Group " + group.getId(),e);
017   }
018  }
019  @Override
020  public void insert(Group group) throws MapperException {
021   try {
022    GroupTDG.insert(group.getId(), group.getVersion(), group.getName());
023   } catch (SQLException e) {
024    throw new MapperException("Could not insert Group " + group.getId(),e);
025   }
026  }
027
028  @Override
029  public void update(Group group) throws MapperException {
030   try {
031    int count = GroupTDG.update(group.getId(), group.getVersion(), group.getName());
032    if(count == 0) throw new LostUpdateException("GroupTDG: id " + group.getId() +
033     " version " + group.getVersion());
034    group.setVersion(group.getVersion()+1);
035   } catch (SQLException e) {
036    throw new MapperException("Could not update Sponsor " + group.getId(),e);
037   }
038  }
039 }
```

Figure 9-11 GroupOutputMapper

```
001 package dom.model.group.tdg;
002
003 // imports
004
005 public class GroupTDG {
006
007  public static final String BASE_NAME = "Group";
008  public final static String TABLE = DbRegistry.getTablePrefix() + BASE_NAME;
009
010  public final static String CREATE_TABLE =
011   "CREATE TABLE IF NOT EXISTS " + TABLE + " (" +
012    "id BIGINT,"+
013    "version int,"+
014    "name varchar(128),"+
015    "PRIMARY KEY(id)"+
016    ") ENGINE=InnoDB;";
017
018  public final static String DROP_TABLE =
019   "DROP TABLE IF EXISTS " + TABLE + ";";
020
021  public final static String DELETE_BYID_SQL =
022   "DELETE FROM " + TABLE + " WHERE id=? AND version=?;";
023
024  public final static String INSERT_BYID_SQL =
025   "INSERT INTO " + TABLE + " (id,version,name) values(?,?,?);";
026
027  public final static String UPDATE_BYID_SQL =
028   "UPDATE " + TABLE + " " +
029   "SET version=version+1,name=? WHERE id=? and version=?;";
030
031  public static void createTable() throws SQLException {
032   SQLLogger.processUpdate(DbRegistry.getDbConnection().createStatement(), CREATE_TABLE);
033  }
034
035  public static void dropTable() throws SQLException {
036   SQLLogger.processUpdate(DbRegistry.getDbConnection().createStatement(), DROP_TABLE);
037  }
038
039  public static int insert(long id, long version, String name) throws SQLException
040  {
041   Connection con = DbRegistry.getDbConnection();
042   PreparedStatement ps = con.prepareStatement(INSERT_BYID_SQL);
043   ps.setLong(1,id);
044   ps.setLong(2,version);
045   ps.setString(3, name);
046   int count = SQLLogger.processUpdate(ps);
047   ps.close();
048   return count;
049  }
050
051  public static int update(long id, long version, String name) throws SQLException
052  {
053   Connection con = DbRegistry.getDbConnection();
054   PreparedStatement ps = con.prepareStatement(UPDATE_BYID_SQL);
055   ps.setString(1, name);
056   ps.setLong(2,id);
057   ps.setLong(3,version);
058   int count = SQLLogger.processUpdate(ps);
059   ps.close();
060   return count;
061  }
062
063  public static int delete(long id, long version) throws SQLException
064  {
065   Connection con = DbRegistry.getDbConnection();
```

```
001 package dom.model.group.tdg;
002
003 // imports
004
005 public class GroupFinder {
006
007  public static String SELECT_BY_ID_SQL =
008   "SELECT g.id,g.version,g.name FROM " + GroupTDG.TABLE + " AS g "+
009   "WHERE g.id=?;";
010
011  public static String SELECT_ALL_SQL =
012   "SELECT g.id FROM " + GroupTDG.TABLE + " AS g;";
013
014  public static String SELECT_BY_NAME_SQL =
015   "SELECT g.id,g.version,g.name FROM " + GroupTDG.TABLE + " AS g "+
016   "WHERE g.name=?;";
017
018
019  public static ResultSet find(long id) throws SQLException{
020   Connection con = DbRegistry.getDbConnection();
021   PreparedStatement ps = con.prepareStatement(SELECT_BY_ID_SQL);
022   ps.setLong(1, id);
023   return SQLLogger.processQuery(ps);
024  }
025
026  public static ResultSet findAll() throws SQLException{
027   Connection con = DbRegistry.getDbConnection();
028   PreparedStatement ps = con.prepareStatement(SELECT_ALL_SQL);
029   return SQLLogger.processQuery(ps);
030  }
031
032  public static ResultSet findByName(String name) throws SQLException{
033   Connection con = DbRegistry.getDbConnection();
034   PreparedStatement ps = con.prepareStatement(SELECT_BY_NAME_SQL);
035   ps.setString(1, name);
036   return SQLLogger.processQuery(ps);
037  }
038 }
```

Figure 9-13 GroupFinder

```
001 package dom.model.groupmembership;
002
003 // imports
004
005 public interface IGroupMembership extends IDomainObject<Long>{
006
007  public abstract IUser getMember();
008
009  public abstract void setMember(IUser member);
010
011  public abstract IGroup getGroup();
012
013  public abstract void setGroup(IGroup group);
014
015  public abstract MembershipStatus getStatus();
016
017  public abstract void setStatus(MembershipStatus status);
018
019  public abstract Calendar getLastUpdated();
020
021  public abstract void setLastUpdated(Calendar lastUpdated);
022
023 }
```

Figure 9-14 IGroupMembership

```
001 package dom.model.groupmembership;
002
003 // imports
004
005 public class GroupMembership extends DomainObject<Long> implements
006   IGroupMembership {
007  private IUser member;
008  private IGroup group;
009  private MembershipStatus status;
010  private Calendar lastUpdated;
011
012  public GroupMembership(Long id, long version, IUser member, IGroup group,
013    MembershipStatus status, Calendar lastUpdated) {
014   super(id, version);
015   this.member = member;
016   this.group = group;
017   this.status = status;
018   this.lastUpdated = lastUpdated;
019  }
020
021  public IUser getMember() {
022   return member;
023  }
024
025  public void setMember(IUser member) {
026   this.member = member;
027  }
028
029  public IGroup getGroup() {
030   return group;
031  }
032
033  public void setGroup(IGroup group) {
034   this.group = group;
035  }
036
037  public MembershipStatus getStatus() {
038   return status;
039  }
040
041  public void setStatus(MembershipStatus status) {
042   this.status = status;
043  }
044
045  public Calendar getLastUpdated() {
046   return lastUpdated;
047  }
048
049  public void setLastUpdated(Calendar lastUpdated) {
050   this.lastUpdated = lastUpdated;
051  }
052 }
```

Figure 9-15 GroupMembership

```
001 package dom.model.groupmembership;
002
003 // imports
004
005 public class GroupMembershipProxy extends
006   DomainObjectProxy<Long, GroupMembership> implements IGroupMembership {
007
008  public GroupMembershipProxy(Long id) {
009   super(id);
010  }
011
012  @Override
013  protected GroupMembership getFromMapper(Long id) throws SQLException,
014    DomainObjectCreationException {
015   try {
016    return GroupMembershipInputMapper.find(id);
017   } catch (MapperException e) {
018    throw new DomainObjectCreationException(e.getMessage(), e);
019   }
020  }
021
022  @Override
023  public IGroup getGroup() {
024   return getInnerObject().getGroup();
025  }
026
027  @Override
028  public Calendar getLastUpdated() {
029   return getInnerObject().getLastUpdated();
030  }
031
032  @Override
033  public IUser getMember() {
034   return getInnerObject().getMember();
035  }
036
037  @Override
038  public MembershipStatus getStatus() {
039   return getInnerObject().getStatus();
040  }
041
042  @Override
043  public void setGroup(IGroup group) {
044   getInnerObject().setGroup(group);
045  }
046
047  @Override
048  public void setLastUpdated(Calendar lastUpdated) {
049   getInnerObject().setLastUpdated(lastUpdated);
050  }
051
052  @Override
053  public void setMember(IUser member) {
054   getInnerObject().setMember(member);
055  }
056
057  @Override
058  public void setStatus(MembershipStatus status) {
059   getInnerObject().setStatus(status);
060  }
061
062 }
```


Figure 9-16 GroupMembershipProxy

```
001 package dom.model.groupmembership;
002
003 // imports
004
005 public class GroupMembershipFactory {
006
007  public static GroupMembership createNew(IUser member, IGroup group,
008    MembershipStatus status, Calendar lastUpdated) throws SQLException {
009   GroupMembership result = new GroupMembership(
010     GroupMembershipTDG.maxId(), 0l, member, group, status,
011     lastUpdated);
012   UoW.getCurrent().registerNew(result);
013   return result;
014  }
015
016  public static GroupMembership createClean(long id, long version,
017    IUser member, IGroup group, MembershipStatus status,
018    Calendar lastUpdated) throws SQLException {
019   GroupMembership result = new GroupMembership(id, version, member,
020     group, status, lastUpdated);
021   UoW.getCurrent().registerClean(result);
022   return result;
023  }
024 }
```

Figure 9-17 GroupMembershipFactory

```
001 package dom.model.groupmembership;
002
003 public enum MembershipStatus {
004  LEADER,
005  INVITED,
006  ACCEPTED
007 }
```

Figure 9-18 MembershipStatus

```
001 package dom.model.groupmembership;
002
003 // imports
004
005 public class MembershipListProxy extends ListProxy<IGroupMembership> {
006
007  private IGroup parent;
008
009  public MembershipListProxy(IGroup parent) {
010   super();
011   this.parent = parent;
012  }
013
014  @Override
015  protected List<IGroupMembership> getActualList() throws Exception {
016   return GroupMembershipInputMapper.find(parent);
017  }
018 }
```

Figure 9-19 MembershipListProxy

```
001 package dom.model.groupmembership.mappers;
002
003 // imports
004
005 public class GroupMembershipInputMapper {
006
007  public static List<IGroupMembership> buildCollection(ResultSet rs)
008    throws SQLException, MapperException, DomainObjectCreationException {
009   return buildCollection(rs, "gm.id");
010  }
011
012  public static List<IGroupMembership> buildCollection(ResultSet rs,
013    String idString) throws SQLException, MapperException,
014    DomainObjectCreationException {
015   ArrayList<IGroupMembership> l = new ArrayList<IGroupMembership>();
016   while (rs.next()) {
017    l.add(new GroupMembershipProxy(rs.getLong(idString)));
018   }
019   return l;
020  }
021
022  public static List<IGroupMembership> findAll() throws SQLException,
023    MapperException, DomainObjectCreationException {
024   ResultSet rs = GroupMembershipFinder.findAll();
025   return buildCollection(rs);
026  }
027
028  public static List<IGroupMembership> find(IGroup myGroup)
029    throws SQLException, MapperException, DomainObjectCreationException {
030   ResultSet rs = GroupMembershipFinder.findByGroup(myGroup.getId());
031   return buildCollection(rs);
032  }
033
034  public static GroupMembership find(long id) throws SQLException,
035    MapperException, DomainObjectCreationException {
036   try {
037    return IdentityMap.get(id, GroupMembership.class);
038   } catch (DomainObjectNotFoundException e) {
039   } catch (ObjectRemovedException e) {
040
041   }
042   ResultSet rs = GroupMembershipFinder.find(id);
043   if (!rs.next())
044    throw new MapperException(
045      "The record for this GroupMembership id doesn't exist");
046   return getGroupMembership(rs);
047  }
048
049  private static GroupMembership getGroupMembership(ResultSet rs)
050    throws SQLException, MapperException, DomainObjectCreationException {
051   Calendar cal = Calendar.getInstance();
052   cal.setTimeInMillis(rs.getLong("gm.lastUpdated"));
053   GroupMembership result = GroupMembershipFactory.createClean(rs
054     .getLong("gm.id"), rs.getLong("gm.version"), new UserProxy(rs
055     .getLong("gm.member")), new GroupProxy(rs.getLong("gm._group")),
056     MembershipStatus.values()[rs.getInt("gm.status")], cal);
057   return result;
058  }
059
060 }
```

Figure 9-20 GroupMembershipInputMapper

```
001 package dom.model.groupmembership.mappers;
002
003 // imports
004
005 public class GroupMembershipOutputMapper implements
006   GenericOutputMapper<Long, GroupMembership> {
007
008  @Override
009  public void delete(GroupMembership membership) throws MapperException {
010   try {
011    int count = GroupMembershipTDG.delete(membership.getId(),
012      membership.getVersion());
013    if (count == 0)
014     throw new LostUpdateException("GroupMembershipTDG: id "
015       + membership.getId() + " version "
016       + membership.getVersion());
017    membership.setVersion(membership.getVersion() + 1);
018   } catch (SQLException e) {
019    throw new MapperException("Could not delete GroupMembership "
020      + membership.getId(), e);
021   }
022  }
023
024  @Override
025  public void insert(GroupMembership membership) throws MapperException {
026   try {
027    GroupMembershipTDG.insert(membership.getId(), membership
028      .getVersion(), membership.getMember().getId(), membership
029      .getGroup().getId(), membership.getStatus().ordinal(),
030      membership.getLastUpdated().getTimeInMillis());
031   } catch (SQLException e) {
032    throw new MapperException("Could not insert GroupMembership "
033      + membership.getId(), e);
034   }
035  }
036
037  @Override
038  public void update(GroupMembership membership) throws MapperException {
039   try {
040    int count = GroupMembershipTDG.update(membership.getId(),
041      membership.getVersion(), membership.getMember().getId(),
042      membership.getGroup().getId(), membership.getStatus()
043        .ordinal(), membership.getLastUpdated()
044        .getTimeInMillis());
045    if (count == 0)
046     throw new LostUpdateException("GroupMembershipTDG: id "
047       + membership.getId() + " version "
048       + membership.getVersion());
049    membership.setVersion(membership.getVersion() + 1);
050   } catch (SQLException e) {
051    throw new MapperException("Could not update GroupMembership "
052      + membership.getId(), e);
053   }
054  }
055
056 }
```

Figure 9-21 GroupMembershipOutputMapper

```
001 package dom.model.groupmembership.tdg;
002
003 // imports
004
005 public class GroupMembershipTDG {
006
007  public static final String BASE_NAME = "GroupMembership";
008  public final static String TABLE = DbRegistry.getTablePrefix() + BASE_NAME;
009
010  public final static String CREATE_TABLE = "CREATE TABLE IF NOT EXISTS "
011    + TABLE + " (" + "id BIGINT," + "version int," + "member BIGINT,"
012    + "_group BIGINT," + "status int," + "lastUpdated BIGINT,"
013    + "PRIMARY KEY(id)," + "INDEX (_group)" + ") ENGINE=InnoDB;";
014
015  public final static String DROP_TABLE = "DROP TABLE IF EXISTS " + TABLE
016    + ";";
017
018  public final static String DELETE_BYID_SQL = "DELETE FROM " + TABLE
019    + " WHERE id=? AND version=?;";
020
021  public final static String INSERT_BYID_SQL = "INSERT INTO "
022    + TABLE
023    + " (id,version,member,_group,status,lastUpdated) values(?,?,?,?,?,?);";
024
025  public final static String UPDATE_BYID_SQL = "UPDATE "
026    + TABLE
027    + " "
028    + "SET version=version+1,member=?,_group=?,status=?,lastUpdated=? WHERE id=? and version=?;"
029
030  public static void createTable() throws SQLException {
031   SQLLogger.processUpdate(DbRegistry.getDbConnection().createStatement(),
032     CREATE_TABLE);
033  }
034
035  public static void dropTable() throws SQLException {
036   SQLLogger.processUpdate(DbRegistry.getDbConnection().createStatement(),
037     DROP_TABLE);
038  }
039
040  public static int insert(long id, long version, Long member, Long group,
041    Integer status, Long lastUpdated) throws SQLException {
042   Connection con = DbRegistry.getDbConnection();
043   PreparedStatement ps = con.prepareStatement(INSERT_BYID_SQL);
044   ps.setLong(1, id);
045   ps.setLong(2, version);
046   ps.setLong(3, member);
047   ps.setLong(4, group);
048   ps.setInt(5, status);
049   ps.setLong(6, lastUpdated);
050   int count = SQLLogger.processUpdate(ps);
051   ps.close();
052   return count;
053  }
054
055  public static int update(long id, long version, Long member, Long group,
056    Integer status, Long lastUpdated) throws SQLException {
057   Connection con = DbRegistry.getDbConnection();
058   PreparedStatement ps = con.prepareStatement(UPDATE_BYID_SQL);
059   ps.setLong(1, member);
060   ps.setLong(2, group);
061   ps.setInt(3, status);
062   ps.setLong(4, lastUpdated);
063   ps.setLong(5, id);
064   ps.setLong(6, version);
065   int count = SQLLogger.processUpdate(ps);
```

```
001 package dom.model.groupmembership.tdg;
002
003 // imports
004
005 public class GroupMembershipFinder {
006
007  public static String SELECT_BY_ID_SQL = "SELECT gm.id,gm.version,gm.member,"
008    + "gm._group,gm.status,gm.lastUpdated FROM "
009    + GroupMembershipTDG.TABLE + " AS gm " + "WHERE gm.id=?;";
010
011  public static String SELECT_ALL_SQL = "SELECT gm.id FROM "
012    + GroupMembershipTDG.TABLE + " AS gm;";
013
014  public static String SELECT_BY_GROUP_SQL = "SELECT gm.id,gm.member FROM "
015    + GroupMembershipTDG.TABLE + " AS gm " + "WHERE gm._group=?";
016
017  public static ResultSet find(long id) throws SQLException {
018   Connection con = DbRegistry.getDbConnection();
019   PreparedStatement ps = con.prepareStatement(SELECT_BY_ID_SQL);
020   ps.setLong(1, id);
021   return SQLLogger.processQuery(ps);
022  }
023
024  public static ResultSet findAll() throws SQLException {
025   Connection con = DbRegistry.getDbConnection();
026   PreparedStatement ps = con.prepareStatement(SELECT_ALL_SQL);
027   return SQLLogger.processQuery(ps);
028  }
029
030  public static ResultSet findByGroup(Long group) throws SQLException {
031   Connection con = DbRegistry.getDbConnection();
032   PreparedStatement ps = con.prepareStatement(SELECT_ALL_SQL);
033   ps.setLong(1, group);
034   return SQLLogger.processQuery(ps);
035  }
036 }
```

Figure 9-23 GroupMembershipFinder

```
001 package dom.model.member.mappers;
002
003 // imports
004
005 public class MemberInputMapper {
006  public static List<IUser> buildCollection(ResultSet rs, String idString)
007   throws SQLException, MapperException, DomainObjectCreationException {
008   ArrayList<IUser> l = new ArrayList<IUser>();
009   while(rs.next()) {
010    l.add(new UserProxy(rs.getLong(idString)));
011   }
012   return l;
013  }
014
015  public static List<IUser> find(IGroup myGroup) throws SQLException,
016   MapperException, DomainObjectCreationException {
017   ResultSet rs = GroupMembershipFinder.findByGroup(myGroup.getId());
018   return buildCollection(rs, "gm.member");
019  }
020 }
```

Figure 9-24 MemberInputMapper

```
001 package dom.model.role;
002
003 // imports
004
005 public class AdminRole extends Role implements IRole {
006
007  private static final long ROLE_ID = RoleIds.ADMIN;
008  private static final String ROLE_NAME = "AdminRole";
009
010  public AdminRole() {
011   super(ROLE_ID, 1, ROLE_NAME);
012  }
013
014  @Override
015  public String getName() {
016   return ROLE_NAME;
017  }
018
019  @Override
020  public Long getId() {
021   return ROLE_ID;
022  }
023
024  @Override
025  public long getVersion() {
026   return 1;
027  }
028
029 }
030
```

Figure 9-25 AdminRole

```
001 package dom.model.role;
002
003 // imports
004
005 public class RegisteredRole extends Role implements IRole {
006
007  private static final long ROLE_ID = RoleIds.ADMIN;
008  private static final String ROLE_NAME = "RegisteredRole";
009
010  public RegisteredRole() {
011   super(ROLE_ID, 1, ROLE_NAME);
012  }
013
014  @Override
015  public String getName() {
016   return ROLE_NAME;
017  }
018
019  @Override
020  public Long getId() {
021   return ROLE_ID;
022  }
023
024  @Override
025  public long getVersion() {
026   return 1;
027  }
028
029 }
030
```

Figure 9-26 RegisteredRole

```
001 package dom.model.role;
002
003 public class RoleIds {
004  public static final long REGISTERED = 2L;
005  public static final long ADMIN      = 3L;
006 }
```

Figure 9-27 RoleIds

```
001 package ts;
002
003 // imports
004
005 public abstract class DatabaseSetup {
006
007  public static void main(String[] args) {
008   setupLogging();
009   FrontController.prepareDbRegistry();
010   FrontController.setupUoW();
011   dropAllTables();
012   createAllTables();
013  }
014
015  public static void setup()
016  {
017   setupLogging();
```

```
018   FrontController.prepareDbRegistry();
019   FrontController.setupUoW();
020   try {
021    startTransaction();
022    createAllTablesNoCommit();
023    finishTransaction();
024   } catch (Exception e)
025   {
026    e.printStackTrace();
027    System.exit(1);
028   }
029  }
030
031  public static void teardown()
032  {
033   try {
034    startTransaction();
035    dropAllTablesNoCommit();
036    finishTransaction();
037    DbRegistry.closeDbConnectionIfNeeded();
038   } catch (Exception e) {
039    e.printStackTrace();
040   }
041  }
042
043  public static void createAllTablesNoCommit() throws SQLException, IOException {
044   createAllTablesNoCommit(true);
045  }
046
047  public static void createAllTablesNoCommit(boolean doBaseDataInsert) throws SQLException,
IOException
048  {
049   UserTDG.createTable();
050   UserTDG.createUserRoleTable();
051   GroupTDG.createTable();
052   GroupMembershipTDG.createTable();
053   List<IRole> roles = new ArrayList<IRole>();
054   roles.add(new AdminRole());
055   roles.add(new GuestRole());
056   roles.add(new RegisteredRole());
057   UserFactory.createNew("sthiel", "sthiel", roles);
058
059  }
060
061  public static void createAllTables() {
062   createAllTables(true);
063  }
064
065  public static void createAllTables(boolean doCars) {
066   FrontController.setupUoW();
067
068   try {
069    startTransaction();
070    createAllTablesNoCommit(doCars);
071    UoW.getCurrent().commit();
072   } catch (Exception e) {
073    e.printStackTrace();
074   } finally {
```

```
075    try {
076     DbRegistry.closeDbConnectionIfNeeded();
077    } catch (Exception e) {
078     e.printStackTrace();
079    }
080   }
081  }
082
083  public static void dropAllTablesNoCommit() throws SQLException
084  {
085   try {
086    UserTDG.dropTable();
087   } catch (Exception e) { }
088   try {
089    UserTDG.dropUserRoleTable();
090   } catch (Exception e) { }
091   try {
092    GroupTDG.dropTable();
093   } catch (Exception e) { }
094   try {
095    GroupMembershipTDG.dropTable();
096   } catch (Exception e) { }
097
098  }
099
100
101  public static void dropAllTables() {
102
103   try {
104    DbRegistry.getDbConnection().setAutoCommit(false);
105    dropAllTablesNoCommit();
106    DbRegistry.getDbConnection().commit();
107   } catch (Exception e) {
108    e.printStackTrace();
109   } finally {
110    try {
111     DbRegistry.closeDbConnectionIfNeeded();
112    } catch (Exception e) {
113     // TODO Auto-generated catch block
114     e.printStackTrace();
115    }
116   }
117  }
118
119
120  public static void startTransaction() throws SQLException {
121   DbRegistry.getDbConnection().setAutoCommit(false);
122   DbRegistry.getDbConnection().createStatement().execute("Start Transaction");
123   UoW.newCurrent();
124  }
125
126  public static void finishTransaction()
127  {
128   try {
129    UoW.getCurrent().commit();
130   }
131   catch (Exception e) {
132    e.printStackTrace();
```

```
133   }
134  }
135
136  public static void setupLogging() {
137
138      String loglevel = null;
139   try {
140    loglevel = Registry.getProperty("LogLevel");
141    if(loglevel.trim().equals("")) throw new Exception("EmptyProperty");
142   } catch (Exception e1) {
143    loglevel = "INFO";
144   }
145
146      try {
147       for(Handler h : Logger.getLogger(FrontController.LOG_STRING).getHandlers()) {
148        Logger.getLogger(FrontController.LOG_STRING).removeHandler(h);
149       }
150
151    ConsoleHandler ch = new ConsoleHandler();
152    ch.setLevel(Level.ALL);
153    Logger.getLogger(FrontController.LOG_STRING).addHandler(ch);
154    Logger.getLogger(FrontController.LOG_STRING).setLevel(Level.parse(loglevel));
155   } catch (SecurityException e) {
156    e.printStackTrace();
157   }
158
159
160  }
161 }
```

Figure 9-28 DatabaseSetup

```
001 <access>
002  <role name="org.dsrg.soenea.domain.role.impl.GuestRole">
003   <command name="app.dispatcher.LoginDispatcher" get="true" post="true" />
004   <command name="app.dispatcher.LogoutDispatcher" get="true" post="true" />
005  </role>
006
007  <role name="dom.model.role.RegisteredRole">
008   <command name="app.dispatcher.CreateGroupDispatcher" post="true" />
009   <command name="app.dispatcher.RemoveGroupDispatcher" post="true" />
010   <command name="app.dispatcher.ViewGroupDispatcher" get="true"/>
011   <command name="app.dispatcher.InviteMemberDispatcher" post="true" />
012   <command name="app.dispatcher.AcceptInviteDispatcher" post="true" />
013   <command name="app.dispatcher.RemoveMemberDispatcher" post="true" />
014  </role>
015
016  <role name="dom.model.role.AdminRole">
017   <command name="app.dispatcher.AddUserDispatcher" post="true" />
018  </role>
019 </access>
```

Figure 9-29 Access.xml

```
001 mySqlHostName=localhost:3306
002 mySqlUserName=soenea
003 mySqlPassword=soenea
004 mySqlDatabase=soenea?characterEncoding=utf8
005 mySqlTablePrefix=soenea_
006 LogLevel=ALL
007 LogFile=
008 myDefaultServletEntrypoint=index.html
009 AccessXMLFile=WEB-INF/classes/Access.xml
010 ConcreteRole_1=org.dsrg.soenea.domain.role.impl.GuestRole
011 ConcreteRole_2=dom.model.role.RegisteredRole
012 ConcreteRole_3=dom.model.role.AdminRole
013 defaultDispatcher=app.dispatcher.LoginDispatcher
014 GUEST_USER_ID=-1
```

Figure 9-30 MyResources.properties