15. Architectural Refactoring

Iteration 1 of the EM case-study resulted in a running executable that could be demonstrated to the project customers and other stakeholders. Clear and robust PCMEF architecture (Chapter 9) is a special strength of Iteration 1 deliverable. Consequently, the software is supportable, i.e. it is understandable, maintainable, and scalable.

It is fair to say, however, that of these three features, Iteration 1 emphasizes understandability, represented in the minimization of object coupling. Minimization of object coupling can easily lead to low cohesion of objects and, therefore, to difficulties with maintainability and scalability. There is a need for a better balance between all three features. This better balance can be achieved by refactoring.

“Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure” (Fowler, 1999, p.XVI). Refactoring is about cleaning up the code after it has been written. Refactoring targets are potential problem areas in the design so far. Fowler calls these problem areas “bad smells in code”.

Refactoring integrates very well with the agile development (Martin, 2003). It can be conducted at any point during iteration, but it is most effective towards the end of the current iteration or at the beginning of the next iteration. Refactoring can significantly improve the internal structure of the code without changing its external behavior.

It can be argued that the practice of test-driven development (Section 12.2) is a partial substitution for refactoring. In reality, test-driven development uses a variation of refactoring – a variation that applies to cleaning up the design rather than the code. Test-driven development is an iterative and incremental process intermixed with writing the application code. Refactoring can anticipate “bad smells in code” and eliminate them before they happen.

In here, refactoring is used at the beginning of Iteration 2 in order to restructure Iteration 1 code for maintainability and scalability, and to prepare it for the Iteration 2 increment. The main changes will be in the domain layer; less so in the control and foundation layers. The presentation layer is not refactored because Iteration 2 effectively replaces Iteration 1 presentation (and any refactoring would be immediately lost).

Note also, that the development and code of Iteration 1 is suboptimal for pedagogically-motivated reasons. Many important object-oriented technologies were not used in Iteration 1 because of the book’s pedagogy and the adopted sequence of topic presentation. Accordingly, the discussion of refactoring in this chapter is suboptimal as well.
15.1 Refactoring Targets

Fowler (1999) identifies and names the whole range of “bad smells in code”. This is one kind of refactoring targets. Other targets are not that much the result of “bad smells”, but the result of a desire to further improve the code, in particular to improve the architecture of the code (Fowler, 2003).

Some more interesting refactoring targets triggered by “bad smells” are:

- duplicated code – the same pieces of code in multiple places,
- long method – a method that does too much,
- large class – a class that does too much and/or has too many data members,
- long parameter list – too much data passed in parameters (rather than asking other objects for the data),
- divergent change – when a class has to be changed as a result of more than one kind of change,
- shotgun surgery – when the same change affects many classes,
- feature envy – a method that accesses many other objects with get messages in order to get data for its own computation,
- data clumps – data items (data members, parameters) that tend to be used together in many places and should be made into an object.

A refactoring target has a number of refactoring methods or refactoring patterns to eliminate “bad smells” or to introduce architectural improvements. Sometimes, the same refactoring method or refactoring pattern can be used to resolve (or partly resolve) more than one refactoring target.

15.2 Refactoring Methods

In the software lifecycle, the time and effort put into the code maintenance significantly outweighs the time and effort put into writing the code in the first place. Code maintenance is reading it and trying to understand it in order to modify or extend it. Any refactoring of code during its production, no matter how small, can significantly benefit software maintainers.

Refactoring methods (or simply refactoring) are basic principles and best practices of changing the code to improve its understandability, maintainability and scalability. The code changes are small, one step at a time, but the improvements can be quite dramatic. Unfortunately, so can be deteriorations of code, if refactorings are not done properly.

In contemporary practice, CASE and programming development tools can effectively assist in performing refactorings. Many tools contain catalogs of supported refactorings. Fowler (1999) is a principal source of reference that lists and documents in excess of sixty refactoring methods. The following discussion illustrates the use of refactoring methods by discussing just three of them:

- Extract Class
- Subsume Method
- Extract Interface
15.2.1 Extract Class

One of refactoring targets is called large class – a class that does too much and/or has too many data members. Large classes can result from excessive minimization of coupling between classes. In case of the large class target, two methods relevant to Iteration 1 code are: Extract Class and Extract Interface (Fowler, 1999).

The Extract Class refactoring is defined as “Create a new class and move the relevant fields and methods from the old class into the new class” (Fowler, 1999, p.149). The main difficulty is in deciding how to split a large class into a number of smaller classes. The idea is to extract consistent and integrated piece of functionality into a separate class (classes).

Once fields and methods are relocated to new class (classes), an association link should be established from the old to the new class. This can be one-way link unless there is an obvious need for backward message passing. Of course, the relocation of methods has an impact on client classes that depended on these methods. This necessitates changes to method invocations in these classes.

Figure 15-1 shows how the Extract Class refactoring could be applied to the CActioner class (Section 13.4.1). CActioner is involved in two quite disparate tasks: in retrieving outmessages requested by the user and in sending (emailing) outmessages. It is logical to extract these two tasks into separate classes: CMsgSeeker and CMsgSender. To avoid terminological confusion, CActioner is renamed to CAdmin. Constructor and non-public methods are not considered.

As expected by the Extract Class refactoring, CAdmin maintains association links to new classes. The links are supported by two methods: getMsgSeeker() and getMsgSender(). The former gets the
CMsgSeeker object, the one responsible for retrieving outmessages for presentation layer. The latter gets the CMsgSender object responsible for emailing outmessages.

Refactoring results in one new method (setEmployee()). setEmployee() is used to establish an association from CMsgSeeker to EEmployee. This is required because outmessage retrievals require employee's data. setEmployee() gets an EEmployee object from CAdmin’s login() method.

15.2.2 Subsume Method

Duplicated code (the same pieces of code in multiple places) is a frequent refactoring target. Depending on the nature of it, duplicated code can be addressed in a number of different ways. A typical refactoring for duplicated code is Extract Method, i.e. turning code duplicated in several methods into a separate method (Fowler, 1999). Another suitable refactoring, not listed by Fowler, can be Subsume Method. The Subsume Method refactoring eliminates a method by including its functionality into another existing method.

The Extract Class refactoring in Figure 15-1 reveals that a CActioner’s method retrieveMessage() is not present in the refactored code. The elimination of retrieveMessage() is not due to Extract Class. It is due to Subsume Method that combines retrieveMessage() into retrieveMessages().

The Iteration 1 code is unduly influenced by two separate menu options available to the user: View Unsent Messages and Display Text of a Message (Figure 12-12). The first option results in retrieving and displaying many outmessages, the second – in retrieving and displaying one outmessage that the user intends to email. In both cases, complete data content of EOutMessage objects is retrieved but the display of information to the user varies.

The truth of the matter is that there is lots of duplicated code resulting from independent processing of these two menu actions, as observed in Section 13.6.1.4. Surely, retrieving many outmessages and retrieving a single outmessage involves very similar processing. Figure 15-2 demonstrates the two sequences of message passing for Iteration 1 and how they are combined into a single path starting from the control layer. The distinction between many outmessages and a single outmessage is only evident in the presentation layer.
Figure 15-2: Subsume Method refactoring
15.2.3 Extract Interface

The refactoring target of *Extract Interface* is twofold and defined as “Several clients use the same subset of a class’s interface, or two classes have part of their interfaces in common” (Fowler, 1999, p.341). The *Extract Interface* refactoring method is to “Extract the subset into an interface” (Fowler, 1999, p.341). The idea of this refactoring is related to the very nature of interfaces (Section 9.1.6).

Like *Extract Class* (Section 15.2.1), *Extract Interface* is frequently a response to a large class target. This is a recurrent refactoring target in Iteration 1. PConsole is a large class that does too much. Depending on a menu option selected by the user, PConsole performs distinct tasks. For example, it may need to display a list of outmessages or details of a single outmessage.

In Iteration 1, the execution of these different tasks is encapsulated by the method `displayMenu()` (Section 13.3.2.2, Listing 13-16). This method contains a private method `getUserInput()`, which accepts user’s request and decides on a task by invoking appropriate PConsole method, such as `displayMessages()`. PConsole has only two public methods: `displayLogin()` and `displayMenu()`. Everything else is encapsulated. Encapsulation has advantages but “smells” of a large class that may be uneasy to maintain and scale.

Refactoring of PConsole can first use the *Extract Class* method to create, for example, two separate classes to handle the display of a list of outmessages and the display of a single outmessage (Figure 15-3). The two new classes POutMsgListViewer and POutMsgBodyBuilder take pertinent responsibilities from PConsole (renamed now to PMenu).
A natural next step is to acknowledge that two new presentation classes in the refactored code use the same `display()` method to display a string to screen. Clearly, the code will gain in readability and documentability if the `display()` method is extracted into an interface, implemented by `PMenu` and used, as needed, by the new presentation classes.

Figure 15-3 represents design refactored by using *Extract Class* and *Extract Interface* on the presentation layer. Refactoring of this layer has limited value for Iteration 2. The move in Iteration 2 to graphical user interface signifies quite radical re-organization of the presentation layer. Even so, the existence of interface and new classes will facilitate the transition. In some way, more radical refactoring would be justified. For example, `POutMsgBodyBuilder` can be split into two classes to separate displaying of outmessage from preparing it for emailing.

### 15.3 Refactoring Patterns

Refactoring is inseparable from architectural design and framework development. Significant architectural improvements can be achieved by coordinated application of multiple refactoring methods. Initial system
design is guided by architectural patterns, as discussed in Section 9.3. Architectural patterns are also instrumental for refactorings aimed at more significant code corrections or extensions.

**Refactoring patterns** are architectural patterns used in code refactoring. Fowler (2003) calls them patterns of enterprise application architecture. They target those aspects of the system that make it an enterprise application, as opposed to small-scale desktop applications. They address such issues as communication with the database, in-memory caches of persistent objects, transaction and concurrency management, web presentation, working with distributed objects, etc.

Refactoring patterns position the system for easy scalability when it grows to a large-scale solution. Iteration 1 of the EM case-study emphasizes the database presence in the application while retaining quite primitive text-based user interface. Iteration 2 makes a move to a graphical user interface. Refactoring patterns discussed in this chapter apply still to Iteration 1 and, therefore, concentrate on other than user interface issues.

The following discussion illustrates the architectural patterns used for refactoring of the Iteration 1 code within the domain layer. Most of these patterns are documented in Fowler (2003) – a state-of-the-art source of reference that lists and documents more than fifty architectural patterns. The patterns are presented in the following groups:

- **Identity Map**
- **Data Mapper**
- **Lazy Load**
- **Unit of Work**

### 15.3.1 Identity Map

Iteration 1 acknowledges the importance of assigning object identifiers (OIDs) to entity objects (Section 13.5) and the importance of maintaining entity caches by the mediator package (Section 13.6). This is good, but currently MBroker “smells” of a large class and a number of refactoring patterns can be used to improve identity management and caching. One such pattern is **Identity Map** (Fowler, 2003).

The *Identity Map* pattern “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). An *Identity Map* object maintains one or more maps (e.g. hash maps) that map an object identifier to an object (Section 12.1.2.2). *Identity Map* has two principal objectives:

- to ensure that the same database record is not loaded many times into different application objects (this would create havoc if the application modified one of these objects, but not the others),
- to avoid unnecessary loading of the same data many times (this would impact on the application’s performance).

Fowler (2003) distinguishes between explicit and generic identity maps. Objects in an **explicit identity map** are accessed (get), registered (put) and unregistered (remove) with distinct methods for each class of cached objects. For example, to access an object:

```java
getEEmployee (new Integer(empOID));
```
Objects in a generic identity map are accessed (get), registered (put) and unregistered (remove) with a single method for all classes. A parameter of the method determines the class. For example, to access an object:

```java
get(“EEmployee”, new Integer(OID));
```

Generic map can only be used if all object identifiers are generated by the same algorithm and are globally unique for all objects across all classes. Generic map allows easy creation and maintenance of a single registry of object identifiers that other objects can use to find an object by its OID.

Explicit map has an advantage of readability and documentability. All available maps are statically stated in the code. Additions and deletions of maps are localized to a single class, so adding and deleting new methods is not a problem.

Fowler (2003) distinguishes further between one map per class and one map per session. One map per session requires globally unique object identifiers. In this sense, it goes hand in hand with a generic identity map. One map per class can use object identifiers unique within a class, but it faces a problem of how to handle classes in an inheritance tree. In both cases, identity maps on updateable objects must be given transactional protection and marked as dirty whenever identity objects get out of sync with the corresponding database records.

In case of Iteration 1, the benefits of the Identity Map pattern, as well as the motivations behind the Extract Class method, give rise to a refactoring that leads to a new class EIdentityMap (Listing 15-1). EIdentityMap defines two maps (Lines 19 – 20). One map maps OIDs to entity objects, the other maps msgID to OID of the EOutMessage. Iteration 1 does not require other maps, but they can be added whenever needed. The example shows methods needed to handle mapping of EContact objects.

```
Listing 15-1 Class EIdentityMap - fragment

```

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>public class EIdentityMap {</td>
</tr>
<tr>
<td>19</td>
<td>private Map OIDToObj; //OID -&gt; Obj</td>
</tr>
<tr>
<td>20</td>
<td>private Map msgPKToOID; //msgFK -&gt; OID</td>
</tr>
<tr>
<td>21</td>
<td>}</td>
</tr>
<tr>
<td>22</td>
<td>public EIdentityMap() {</td>
</tr>
<tr>
<td>23</td>
<td>OIDToObj = new HashMap();</td>
</tr>
<tr>
<td>24</td>
<td>msgPKToOID = new HashMap();</td>
</tr>
<tr>
<td>25</td>
<td>}</td>
</tr>
<tr>
<td>26</td>
<td>public IIdentityMap() {</td>
</tr>
<tr>
<td>27</td>
<td>/* Get the stored contact */</td>
</tr>
<tr>
<td>28</td>
<td>public IContact findContact(int contactOID) {</td>
</tr>
<tr>
<td>29</td>
<td>return (IContact) OIDToObj.get(new Integer(contactOID));</td>
</tr>
<tr>
<td>30</td>
<td>}</td>
</tr>
<tr>
<td>31</td>
<td>public void registerContact(IEObjectID oidObject) {</td>
</tr>
<tr>
<td>32</td>
<td>OIDToObj.put(new Integer(oidObject.getOID()), oidObject);</td>
</tr>
<tr>
<td>33</td>
<td>}</td>
</tr>
<tr>
<td>34</td>
<td>public void unregisterContact(IEObjectID oidContact) {</td>
</tr>
<tr>
<td>35</td>
<td>OIDToObj.remove(new Integer(oidContact.getOID()));</td>
</tr>
<tr>
<td>36</td>
<td>}</td>
</tr>
</tbody>
</table>
Class EIdentityMap - fragment

EIdentityMap is an explicit identity map – it has separate methods for each entity class. The class represents the strategy of one map per session – it uses globally unique OIDs.

15.3.2 Data Mapper

The Data Mapper pattern is defined as “a layer of Mappers that moves data between objects and a database while keeping them independent of each other and the mapper itself.” (Fowler, 2003, p.165). The generic Mapper pattern, referred to in the definition, is “an object that sets up a communication between two independent objects.” (Fowler, 2003, p.473). Mapper differs from the Mediator pattern (Section 9.3.5) in that objects decoupled by a mapper object are blissfully unaware of each other, while objects using mediator are aware of it.

In PCMEF framework, Data Mapper belongs to the mediator package. In Iteration 1, the MBroker class (Section 13.6.1) is responsible, among other things, for knowing of entity objects in the cache and for requesting data from the database if an object is not in the cache or if the cache is dirty. These responsibilities of Data Mapper need to be extracted from MBroker into separate class(es), using the Extract Class refactoring (Section 15.2.1). MDataMapper is one such new class in the refactored Iteration 1 code.

Figure 15-4 is a UML activity diagram that shows how MDataMapper decouples the entity package, (responsible for the cache) from the foundation package (responsible for accessing the database). The activity diagram is used here for the convenience of it rather than for the demonstration of its normal applicability. The notation used is consistent with UML 2.0. The notes explain the meanings of symbols. In UML 2.0, activity diagrams are used exclusively for depicting behavior using a control and data flow model reminiscent of Petri nets (UML, 2002). The old UML 1.4 activity diagrams served a combined purpose of behavior specification and state machine (Maciaszek, 2001).
control package: CAdmin

give me entity object (e.g. EContact)

mediator package: MDataMapper

object

object flow

let's find it

decision

find it by OID

where is it?

[ in cache ]

[ not in cache ]

retrieved from the database

entity package: EIdentityMap

check if clean

[ dirty ]

is it clean?

[ clean ]

get it from the cache

entity package: EContact

foundation package: FReader

control flow

Figure 15-4 Activity diagram for MDataMapper

The semantics of the activity model in Figure 15-4 is easy to follow. The meaning of graphical objects is explained in UML notes. To start with, a control object (CAdmin) requests MDataMapper to obtain an entity object. CAdmin may guide this request in a number of ways:

- CAdmin can know an OID of an object and pass it to MDataMapper, or
- CAdmin can know some attribute values of an object, perhaps the primary key value, and pass this information to MDataMapper as a search condition, or
- CAdmin can request a referential search by asking MDataMapper to find objects that are linked to the object known by CAdmin.

MDataMapper needs to establish where the object(s) is. Is it in memory cache or has to be retrieved from the database? If in memory cache, is it clean? If not, it must be retrieved from the database. EIdentityMap (Listing 15-1) knows if an object is in cache. If EIdentityMap has the objects and they are clean, they will be returned to MDataMapper and then to CAdmin. If not, MDataMapper constructs relevant SQL queries and passes them to FReader for access to the database.
15.3.2.1 Load – Check-out

Figure 15-4 is not a complete account of MDataMapper responsibilities. Once data records are retrieved from the database, MDataMapper makes them into objects (i.e. MDataMapper creates new entity objects). The new objects are then registered in EIdentityMap. This is a load operation – database records are retrieved (loaded) and transformed to memory objects. From the database perspective, the load is a check-out operation (records are checked-out to the program). The load process is also called materialization (Larman, 2002).

A fragment of the sequence diagram in Figure 15-5 illustrates a possible scenario for object loading in the refactored Iteration 1 code. The diagram relates to the “Login” interaction (Section 11.4.1). It shows how MDataMapper retrieves and creates a new EEmployee object when asked to get it by employee name. This operation includes generation of an OID for EEmployee, which is then passed to EIdentityMap in the parameter of registerEmployee().

15.3.2.2 Unload – Check-in

Naturally enough, if Data Mapper is responsible for loading objects from the database, it should also be responsible for the opposite activity of unloading objects to the database. From the database perspective, this is check-in operation. The unload process is also known as passivation and dematerialization (Larman, 2002). The unload operation is required when:

- a new entity object is created by the application and it needs to be persistently stored in the database, or
an entity object gets updated by the application and the changes need to be persistently stored in the database, or

an entity object is deleted by the application and, therefore, the corresponding database record must be deleted from the database.

Unloading is subject to the scrutiny of business transactions and the database commit and rollback operations. The topic of transaction management is discussed in detail in Part D of the book in the context of Iteration 3 of the case-study.

Iteration 1 does not create or delete entity objects, but it does update EOutMessage objects to indicate that they have been emailed. This is the task of the “Email Message” interaction (Section 11.4.5). A fragment of the sequence diagram in Figure 15-6 illustrates how an “update”unload can be implemented in the refactored Iteration 1 code.

CMsgSeeker sends an outmessage and it then asks MModerator to updateMessage(). MModerator delegates this task to MDataMapper, which constructs an SQL update statement. FWriter makes the change in the database. This permits MDataMapper to unregisterMessage() in EIdentityMap. Part of this operation is to flagCache() as dirty. The example assumes that MDataMapper has its own map of OIDs.
15.3.3 Alternative Data Mapper Strategies

In the refactored Iteration 1, a single Data Mapper class (MDataMapper) is used to map and unmap raw data into all entity objects known to the application (EEmployee, EContact and EOutMessage). MModerator serves a façade for the mediator package. MDataMapper maintains associations to EIdentityMap on one hand, and to FReader and FWriter on the other hand. This is shown in Figure 15-7.

More complex systems or successive iterations may demand alternative solutions based on various refactoring patterns. The following list pinpoints frequent alternative strategies (Fowler, 2003; Larman, 2002):

1. Many Data Mappers, perhaps one for each entity class.
2. Using metadata and Java reflection to dynamically generate Data Mappers as needed by the application.
3. Lazy check-in that maps only data currently needed by the application and, as one possibility, creates empty objects (proxies) for data not retrieved from the database but linked to the data already retrieved.
15.3.3.1 Many Data Mappers

MDataMapper in the refactored Iteration 1 has separate methods for managing client requests targeting different entity objects. This is needed because there are many different ways in which Data Mapper is asked to find entity objects. Additionally, each entity object has its own unique data members and associations to other objects. In the case of a larger number of entity classes managed by the application, a single Data Mapper can become a bloated class.

Figure 15-8 shows an alternative Iteration 1 solution with many data mappers – one Data Mapper per entity class. The three mappers implement, in their unique ways, the IMDataMapper interface. MModerator maintains one-to-many association to IMDataMapper objects (Larman, 2002).

![Diagram of Many Data Mappers]

The meanings of methods are:

- **getByOID()** – get an entity object if given its OID; retrieve it from database if cache is dirty,
- **retrieve()** – retrieve an entity object from the cache (if it is there) or from the database; if the latter, create a new entity object and put it in the cache,
- **insert()** – convert a new entity object (created by the application) to raw data and insert it into the database,
- **update()** – save the changes to an entity object to the database,
- **delete()** – delete an entity object from the database and remove it from the cache.

Note that if a separate Data Mapper is used for each entity class, then it may be desirable to also have one Identity Map (Section 15.3.1) per entity class. Each mapper maintains then its own cache.

15.3.3.2 Metadata Mapping

The aim of the Metadata Mapping pattern (Fowler, 2003) is to dynamically generate the object-relational mapping based on metadata (data about data). Metadata stores the knowledge about mappings between
tables and classes, columns and data members (fields), foreign keys and associations links, etc. Section 10.2.2 describes the kinds of mappings that can be stored in metadata.

The program can interrogate the metadata at runtime and dynamically construct SQL statements on one hand and getters/setters on entity objects on the other hand. From the scalability perspective, Metadata Mapping permits easy adaptation of the application to database schema changes and to data-definitional changes, in particular additions, of entity classes.

An important decision related to Metadata Mapping is where to keep the metadata. There are three possibilities (Fowler, 2003):

- directly in the source code of the application (in the mediator package in the case of the PCMEF framework),
- in an external file, preferably an XML file (application reads the file during initialization and creates corresponding mappings in the program structure),
- in the database (this allows sharing the same metadata by multiple applications and enables automatic checking for the latest changes of database schema).

There are two main strategies to embed the mapping information into the running code (Fowler, 2003):

- code generation, which reads metadata information and generates classes to do the mapping (this is done in the build process prior to program compilation),
- reflective program, which allows customizing the application to any new metadata information at runtime.

In PCMEF, classes that could do Metadata Mapping belong to the mediator package. Figure 15-9 presents a simple class model for Metadata Mapping. The model is not implemented in the EM case-study.

![Diagram of Metadata Mapping pattern](image-url)
MDataMapper depends on the two metadata classes for finding mapping information necessary to perform its methods. Methods that retrieve the database, such as retrieve(), will ask MTableToClass to get entityClass corresponding to the tableName from which data is being retrieved. Methods that write to the database, such as insert(), will ask MTableToClass to get tableName and MColumnToField to get columnNames.

15.3.4 Lazy Load

Database records and the corresponding entity objects in the application are very interrelated. A request to load an entity object may result in a need to load a long series of interrelated entity objects. These interrelated entity objects correspond to database records linked by referential integrity constraints (i.e. foreign key mappings to primary key records in other tables).

In most cases, an application must not be allowed to try to load all interrelated objects. Firstly, this may not be possible because of the limited size of the memory cache. Secondly, this may not be permissible because of the concurrency requirement of the database, which must allow many applications to use the same records in a conflict-free way. Thirdly, loading objects that may not be needed by the application has a negative impact on performance.

There are two main kinds of retrieval operations to be considered:

- identity load – retrieves specific objects by providing an OID or a primary key value,
- predicate load – retrieves many objects by performing a query search based on predicate conditions, possibly following referential integrity constraints.

There are three main loading strategies, which control how many objects, from the result set returned by a retrieval operation, will be actually stored in the cache:

- closure load – all objects reachable from the specified object are loaded,
- flat load – only specified objects, with no component objects, are loaded,
- n-levels load – objects reachable within given number of levels from the specified object are loaded.

As explained, closure load, called sometime eager load, is normally not acceptable. Remaining strategies are variants of lazy load. The Lazy Load pattern is defined as “an object that doesn’t contain all of the data you need but knows how to get it” (Fowler, 2003, p.200). Some approaches to implement Lazy Load are:

- Lazy Initialization
- Virtual Proxy
- OID Proxy

15.3.4.1 Lazy Initialization

The Lazy Initialization pattern (Beck, 1997) is a simplest variant of the Lazy Load pattern. On request from a client object, Data Mapper – which is responsible for maintaining the entity cache – searches the cache for data and, if the data is not there, loads it from the database.

The implementation of Lazy Initialization differs depending on several factors. Data Mapper’s attempt to obtain data from the cache can be done by looking at the value of a data member in an entity object. If
the value is null (and null is not expected according to business rules), then Data Mapper has to initialize the data member by making a trip to the database. But there may be several complications.

Data member may be a primitive data type, a class reference, or a collection of class references. Each case requires different implementation machinery. Also, a null value can be a legitimate value rather than a placeholder for a missing object yet to be loaded. Finally, the dependency structure of adopted architectural framework may not allow entity objects to depend on foundation objects to get database data (PCMEF uses MDataMapper to solve this problem but there is still the issue of dependency between entity and mediator).

Listing 15-2 demonstrates how Lazy Initialization can be applied within EOutMessage assuming that the application contains a data mapper with contactOID to loaded EContact object and a data member with contactID corresponding to a foreign key in the database. When EOutMessage is asked to getContact(), it checks if contactOID is null. If it is, then EContact is not loaded. To load it, EOutMessage sends a retrieveContact() message to MDataMapper.

Listing 15-2 Method getContact() in EOutMessage

<table>
<thead>
<tr>
<th>Method getContact() in EOutMessage</th>
</tr>
</thead>
<tbody>
<tr>
<td>public IAContact getContact() {</td>
</tr>
<tr>
<td>if (contactOID == null)</td>
</tr>
<tr>
<td>contact = MDataMapper.retrieveContact(contactID);</td>
</tr>
<tr>
<td>return contact;</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

15.3.4.2 Virtual Proxy

The Virtual Proxy pattern (Gamma et al., 1995) is more powerful alternative to Lazy Initialization. Proxy is a placeholder object that stands-in for the real object. It receives messages destined to a real object (called real subject in Virtual Proxy) and creates the real object, if it does not exist yet. In Lazy Load, proxy stands for an object that may need to be loaded on an attempt to access it.

Figures 15-10 and 15-11 show how Virtual Proxy can be used in the context of Iteration 1 code. MDataMapper is responsible for the entity cache. In the EM case-study, entity objects are identified by the application-allocated OIDs. To illustrate Virtual Proxy, let entity objects be identified by their primary keys (PK-s). Assume that EOutMessage has been loaded and that EContactProxy knows the PK of its real subject. When MDataMapper creates EContactProxy, it initializes its contactID field with its corresponding foreign key (FK) value in EOutMessage.

As seen in Figure 15-10, EOutMessage maintains an association to contact. The association is typed with IAContact interface. In reality, the link is to EContactProxy. When MDataMapper asks for contact (getContact()), EOutMessage returns EContactProxy.
When in the next step MDataMapper asks, for example, for contact’s firstName (Figure 15-11), EContactProxy checks if its real subject (EContact) exists. If it does not, it requests MDataMapper to instantiate it. EContactProxy can then return firstName. The next request for familyName, follows the same routine except that EContact is already loaded.
An interface in the entity package, implemented by MDataMapper and used by the proxies, would eliminate the cycle between MDataMapper and EContactProxy (Section 9.1.6.3). In general, there can be many proxies for the same real subject. Making all these proxies to use the same interface will be an extra bonus.

### 15.3.4.3 OID Proxy

The Virtual Proxy pattern is a popular way to implement Lazy Load, but potentially redundant in systems that assign object identifiers (OIDs) to entity objects when loading them. Object programming environments have internal mechanisms of ensuring the identity of objects loaded to memory and use these mechanisms to link objects. The mechanisms vary between languages.

In Java, the mechanism is hidden from the programmer. A link between objects appears to the programmer as containment. For example, if EOutMessage has a field contact, then the programmer
can use this field to access the linked EContact object as if EContact was physically contained in EOutMessage. In C++, the contact would be a memory pointer to EContact.

None of these mechanisms is a replacement for OIDs assigned explicitly by the program to entity objects when they are instantiated in memory. In all but trivial situations, complex transfers of entity objects between the application and the database call for programs to explicitly assign OIDs to in-memory objects. This introduces duplication between internal mechanisms of programming environments to ensure identity of objects and explicit allocation of OIDs.

Duplication is not always bad. The fact that both techniques link in-memory objects can be taken advantage in Lazy Load. The result is that Lazy Load can be fully implemented with proxy-like indirection but without a need for separate proxy classes. A singleton class that maintains maps of OIDs to objects (called EIdentityMap in the refactored Iteration 1 code) replaces proxy classes. The approach can be documented in a pattern – the OID Proxy pattern.

EIdentityMap knows if an entity object is loaded. Upon loading, the object is provided with an OID and all its data members are initialized. Initialization includes foreign key (FK) values obtained from the database. OID-based association links corresponding to foreign keys are initialized to null if the linked object has not been loaded yet.

An additional advantage of OID Proxy over Virtual Proxy is the ease with which the dirty/clean status of an entity object can be determined. When first loaded, the entity object is flagged as clean. If the data content of the object gets out of sync with its corresponding content in the database, the flag is set to dirty. At all times, the entity object knows its status and can trigger re-loading to refresh its content from the database.

There are two variations of OID Proxy that differ in how the program navigates between associated entity objects:
- Navigation in Identity Map
- Navigation in Entity Classes

15.3.4.3.1 Navigation in Identity Map

An Identity Map keeps every loaded entity object in a map. The mapping maintains relationship between an OID and its object. Other maps that relate primary key values to OIDs can also be maintained in order to support requests to search for objects based on their field values. These maps are in general sufficient to navigate between objects that are linked in the database by referential integrity constraints.

The navigation requires that an Identity Map interrogate entity objects to establish association links. An association link in an entity object is represented in two ways:
- by a foreign key (FK) value obtained from the database, and
- by an OID to a related object in memory.

Both FK and OID data members can be null. A null in the foreign key has the same meaning as in the database – the object is not linked. A null in in the OID means that the associated object is not loaded.
Figure 15-12 shows a class model that uses OID Proxy instead of Virtual Proxy. The class EOutMessage contains a data member contactOID that serves as a proxy for the EContact object it links to. As per the Lazy Load pattern, EOutMessage does not contain the data requested when getContact() is called on it, but it knows how to get it.

MDataMapper keeps a list of OIDs of all objects loaded in memory. When passed an OID of any object, it can immediately verify if such object is loaded. When passed a primary key (PK) value of an object (e.g. outmsgID), it can ask EIdentityMap for corresponding OID.

Figure 15-13 explains the sequence of messages for the same scenario as in Figure 15-11. With Identity Map in place, each entity object knows its state (if it is clean or dirty). The state is checked by isDirty(). Assuming that EOutMessage is clean, EIdentityMap asks EOutMessage to getContact(). EOutMessage checks if the foreign key value to EContact is not null (it should not be because in Iteration 1 contactID is not allowed in the database to be null). It then checks if contactOID is not null.
EContact may or may not be loaded in memory. EOutMessage has this knowledge in contactOID. If contactOID is null, then EContact has not been loaded. The sequence diagram in Figure 15-13 assumes that this is the case. EIdentityMap asks MDataMapper to retrieve the EContact based on the foreign key value.

The outcome will be that EContact is loaded (lazily), MDataMapper will update its list of OIDs, EIdentityMap will update its maps, and contactOID in EOutMessage gets set. EContact is now loaded and returned to the client object, which can now ask it directly for data (getFirstName(), getFamilyName(), etc.).
EIdentityMap is in charge of all navigations between entity objects. In this approach, entity objects do not have to contain associated objects. Entity objects are contained in maps of EIdentityMap. Entity objects themselves have only OIDs to other objects.

15.3.4.3.2 Navigation in Entity Classes

Navigation in Identity Map obscures object-relational mapping. Referential integrity constraints are visible in entity objects and they are enhanced (duplicated) by OIDs to linked objects. However, an entity object cannot navigate these links directly. Identity Map is the only place that knows how to get an object given its OID. In a system with large number of entity classes, Identity Map is doing too much.

The responsibility to navigate between entity objects can be placed on these objects. Navigation in entity classes requires that an entity object does not just contain an OID to another object, but that it also contains that object (like in Virtual Proxy example presented in Figure 15-10).

Figure 15-14 presents a sequence diagram, corresponding to Figure 15-13, but which uses navigation in entity classes. Much of the processing logic to findContactForOutmessage() is shifted from MDataMapper and EIdentityMap to EOutMessage.
Upon determining that EOutMessage is clean, MDataMapper instructs to getContact(). If EContact is not loaded, EOutMessage initiates the load. It passes its outmsgOID to MDataMapper so that a new EContact can be associated with it once registered in EIdentityMap.

In the scenario in which a client object asks MDataMapper for the whole EContact, the client object can directly get data from EContact (getFirstName(), getFamilyName(), etc.). However, if EContact has been loaded before, EOutMessage would get the data from EContact (it would also check if EContact is not dirty before obtaining the data).
Navigation in entity classes provides more intuitive and more elegant implementation of the OID Proxy pattern. The semantics of relationships between persistent business objects is implemented directly in these objects. Navigation in identity map requires going back and forth to entity objects to find data in related objects. Navigation in entity objects allows linear movement between entity objects. *Identity Map* is only needed to find initial object from which further navigation can continue.

### 15.3.5 Unit of Work

Entity objects need to be loaded (checked out) from the database for two reasons. Firstly because the application needs them and they have not been loaded. Secondly, because the application finds that they have become dirty (i.e. inconsistent with the database state).

In Iteration 1, the only objects may need to be flagged as dirty after they have been emailed are the EOutMessage objects. Iteration 1 does not make any changes to EContact and EEmployee objects. To simplify things, Iteration 1 makes the whole cache (all entity objects) dirty as soon as any EOutMessage is emailed (Section 13.6.1.7, Listing 13-56). The dirty flag is maintained in the MBroker class.

The refactored Iteration 1 code introduces significant flexibility to cache management. One improvement is the introduction of a dirty flag in each entity object. When loaded, the object is flagged as clean. The flag is changed to dirty when it is modified. This is illustrated in Listing 15-3. After EOutMessage is emailed, MModerator sends a setDirty() message to the affected EOutMessage object (Line 99).

```
Method updateMessage() in MModerator

87:    public boolean updateMessage(IAOutMessage msg) {
91:      msg.setSentDate(new java.sql.Date(System.currentTimeMillis()));
92:      boolean b = mapper.storeMessage(msg);
93:      if (b) {
96:        msg.setContact(null);
97:        msg.setCreatorEmployee(null);
98:        msg.setSenderEmployee(null);
99:        msg.setDirty(true);
100:      }
```

The approach where each entity object knows if it’s dirty or not allows individual loading and unloading of objects. However, many business transactions managed by the application do multiple changes to a variety of objects in the cache before attempting to write these changes to the database. The changed objects need to be marked dirty in unison when the database confirms that the transaction successfully committed. These issues get quickly very complex.

The *Unit of Work* pattern “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).
The changes can be due to three modification operations – inserting a new entity object to the database, deleting an object from the database, or updating of object’s data members.

The *Unit of Work* pattern requires a new class in the mediator package. The class can be called *MWorkUnit* or similar. *Unit of Work* is introduced in Iteration 2, but its full applicability is demonstrated in Iteration 3 (Part D).

### 15.4 Refactored Class Model

The refactored Iteration 1 code consists of increased number of classes as compared with the original design (Section 13.1, Figure 13-1). New classes improve design cohesion, while not impacting adversely on design coupling. Dependencies between classes in the refactored code conform to the PCMEF framework.

Refactored class model is presented here in three class diagrams. The first diagram (Figure 15-15) presents classes in the presentation and control layers. Section 15-2 explains refactorings that resulted in the model in Figure 15-15.
The second diagram for the refactored model (Figure 15-16) concentrates on the domain layer. The notes in the diagram provide basic explanations. The details of refactoring patterns used in the domain layer are discussed in Section 15.3. The use of Java collection interfaces in modeling associations is as discussed in Section 12.1.2.

The foundation layer is the only PCMEF layer that is not affected by refactoring of Iteration 1. However, the dependencies from mediator to foundation change slightly due to replacing MBroker by two classes – MModerator and MDataMapper. This is shown in Figure 15-17.
## Summary

1. **Refactoring** is the process of cleaning up and improving the internal structure of the code without changing its external behavior.

2. There are two main *refactoring targets*: elimination of “bed smells in code” (cleaning up) and structural improvements to the code resulting in a better code architecture.

3. **Refactoring methods** (or simply *refactorings*) are basic principles and best practices of changing the code to improve its supportability (understandability, maintainability and scalability).

4. The *Extract Class* refactoring splits a large class into a number of smaller classes.

5. The *Extract Method* refactoring turns duplicated code into a separate method.

6. The *Subsume Method* refactoring eliminates a method by including its functionality into another existing method.

7. The *Extract Interface* refactoring extracts a set of method signatures duplicated in several classes or used by several clients into an interface.

8. **Refactoring patterns** are architectural patterns used in code refactoring.

9. The *Identity Map* pattern assigns object identifiers to objects and maintains maps to find in-memory objects based on their object identifiers.
10. The Data Mapper pattern decouples in-memory entity objects from their persistent representation in the database and is responsible for maintaining memory caches of entity objects.

11. UML activity diagrams serve as a behavior specification technique able to depict control and data flows.

12. The load (check-out, materialization) is the process of retrieving records from the database and transforming them to memory objects. The opposite operation is called unload (check-out, passivation, dematerialization).

13. The Lazy Load pattern loads only selected objects from database to memory but it can load remaining and related objects when needed. Approaches to Lazy Load are Lazy Initialization, Virtual Proxy, and OID proxy.

14. The Lazy Initialization pattern loads the objects on specific request from a client object responsible for maintaining the entity cache. The load is not transparent to the clients.

15. The Virtual Proxy pattern uses a placeholder object (proxy) that stands-in for the real object and can load the real object in a way that is transparent to the client.

16. The OID Proxy pattern is a handy replacement for the Virtual Proxy pattern in applications that maintain maps of OIDs to objects. The maps can be used instead of proxy classes to load objects in memory in a way that is transparent to the client. There are two variations of OID Proxy that differ in how the program navigates between associated entity objects: Navigation in Identity Map and Navigation in Entity Classes.

17. Navigation in Identity Map uses an Identity Map class every time the application needs to navigate to a linked entity object (i.e. linked by referential integrity). If object X is linked to object Y, Identity Map would interrogate X to obtain the link to Y and it will then access Y.

18. Navigation in Entity Classes uses an Identity Map class to get the first entity object in a navigation sequence but it then relies on entity classes to navigate to related classes. If object X is linked to object Y, Identity Map would allow the client to access X but it will then pass control to X to continue navigation to Y.

19. The Unit of Work pattern makes the application aware of business transactions and concurrency issues. It keeps track of changes to entity objects and whether or not the changes have been committed to the database.

**Key Terms**
15. Architectural Refactoring

OID Proxy pattern ......................................................... 426
passivation ....................................................................... See unload predicate load .......................................................... 422
refactoring ................................................................. 406, See refactoring method, See refactoring method
Refactoring pattern ...................................................... 413, 434
refactoring target ............................................................. 407
Subsume Method refactoring ........................................... 409
Unload pattern ................................................................ 417
Virtual Proxy pattern ..................................................... 423, 435

Review Questions

Discussion Questions

1. Compare the applicability of refactoring in agile development and in Unified Process?

2. Compare refactoring targets of Extract Class and Extract Interface. Use examples, different than in the textbook, to show similarities and differences.

3. The Subsume Method refactoring is not frequently used. Why? Compare it with Extract Method.

4. Explain the objectives of the Identity Map pattern? How is Identity Map related to object-relational mapping?

5. Objects are characterized as having state, behavior and identity. Discuss how well different kinds of Identity Maps support the notion of object identity (OID). Discuss the four kinds of Identity Maps: explicit identity map, generic identity map, one map per session, and one map per class.

6. Discuss advantages and disadvantages of the data mapping strategy based on many Data Mappers.

7. The Metadata Mapping pattern does not specify where the metadata should be kept (stored). What are the possibilities? How do they compare?

8. Provide an example to explain the three lazy load options: closure load, flat load, and n-levels load.


Case-Study Questions

1. Consider the Extract Class refactoring in Figure 15-1. Refer to the setEmployee() method. What would be a likely prototype of this method? (Recall that the method prototype is more than method signature as it includes the return type).

2. Consider the Extract Class refactoring in Figure 15-1 and the Subsume Method refactoring in Figure 15-2. Refer to the retrieveMessages() method. What would be a likely prototype of this method?
3. What needs to be done in control layer if we do not want to expose many of control layer's classes to presentation layer?

4. Suppose that the EM application allows to filter outmessages. This means that the user can apply filter criteria to display only selected subsets of outmessages, e.g. outmessages scheduled for emailing in the future, outmessages scheduled for a particular department, etc. How could we model this?

**Problem-Solving Exercises**

**Case-Study Exercises**

1. Refer to the *Extract Interface* refactoring in Figure 15-3. Could you refactor the code differently? Show the new model.

2. Refer to Case-Study Question 4. Consider that a MessageFilter object is used to implement filtering of outmessages by the application passing the object to appropriate classes in the domain layer. Draw a fragment of sequence diagram that modifies the model for “View Unsent Messages” interaction (Figure 11-12) to include filtering with a MessageFilter object.

3. Most of EM activities are initiated by control layer and delegated to mediator. Mediator further manipulates entity objects as see it fit. We can model this differently by using the Publish-Subscribe pattern on entity objects. When an entity object is modified, it notifies its subscribers about this fact. Present a sequence diagram for this.

4. As shown in Figure 15-4, MDataMapper is responsible for finding entity objects and if necessary load them into memory. Show an alternative design whereby MModerator plays more important role in the process. What refactoring methods and patterns, if any, that drive this modification?