CHAPTER

Data Persistence e13

KEY TOPICS IN THIS CHAPTER

- Object-relational mapping (ORM)
- Virtual proxy
- Brokers
- Virtual caches

13.1 Introduction to data persistence

The availability of persistence mechanisms for commercial languages[1] has made database design much easier for many projects. With adequate tools, it is possible to automatically generate the persistence tier for a great number of information systems. For some critical and legacy systems, however, database adjustments may be still necessary in order to accommodate special features or to satisfy performance or security requirements.

Usually, object-oriented systems are implemented in object-oriented languages, but persistent[2] data storage is accomplished with *relational databases*. Although other techniques such as object-oriented databases (Won, 1990) and XML databases (Bourret, 2010) are also options for implementing permanent storage of data, *ORM*, *object-relational mapping*, is still the preferred approach.

The goal of this chapter is to explain what happens inside the persistence tier of a system when a persistence mechanism based on ORM is used. First of all, a good persistence mechanism requires domain and data storage to be separated into different tiers within the application. Remember that the interface tier is modeled with essential use cases and the domain tier is designed using the conceptual model and contracts as the basis. None of the aforementioned tiers addresses data storage or persistence. Persistence should be designed as a separate concern: a background system that will keep data securely and permanently in its place without interfering on the domain or interface logic.

The persistence mechanism should assure that the objects are saved in a permanent memory device, and that they are loaded from there when necessary. Domain and interface logic should not be polluted by persistence concerns.

Object-oriented design provides lots of good concepts such as encapsulation, responsibility, and delegation that help designers deal with the complexities of the logic for accessing and transforming

[1]See, for example: http://www.hibernate.org/.

[2]*Persistent* in this context is the opposite of *transient*, that is, persistent information is information that must be kept until some user explicitly deletes it. Transient information is kept only during a session of use of the system.

information. But when data must be stored in a more permanent way – disks and tapes, for example – the relational database is a good option because it is very efficient in terms of time performance and there are lots of optimization techniques to improve relational database operations. Thus, if the team wants to work in the best of two worlds (object and relational), the mapping between them must be understood.

The literature refers to the problem of *object-relational impedance mismatch* (Ireland, Keynes, Bowers, Newton, & Waugh, 2009), because most of the good features obtained by object-oriented design are lost when a flat relational database is used.

The implementation of a persistence mechanism minimizes that problem by using a set of predefined classes not belonging to the domain tier that take care of all logic involving domain objects being saved and retrieved from a relational database.

13.2 Object-relational mapping (ORM)

A complete and detailed design class diagram (DCD) allow for the automatic generation of a relational database structure that reflects in secondary memory[3] the information that the objects represent in main memory. The following sections present some equivalence rules that should be observed when using ORM.

13.2.1 Classes and attributes

The first set of rules addresses classes and their attributes. Each persistent class of the DCD corresponds to a *relational table*. Each attribute is a *column* of the table, and each instance is a *line* or *record* of the table.

The stereotypes of some attributes such as «*unique*», «*optional*», and «*immutable*» affect the properties of the columns of the relational tables. Some of the features described here may not be implemented by some commercial database management systems. In that case, adjustments might be necessary in order to provide a safe implementation free from the object-relational impedance mismatch.

Attributes stereotyped with «*unique*» are represented as columns marked as *unique* or *uniq*, which cannot repeat values. However, even objects with no unique attribute have an *identity* that distinguishes them from other objects. Even objects with the same value for every single attribute may be differentiated by their identity. In the case of relational tables, this is accomplished by defining a *primary key* to each relational table. A primary key, just like a unique column, may not repeat elements. An element in a primary key column may also not be null. Primary keys usually are simply sequential numbers generated automatically by the application in such a way that the

[3] *Secondary memory* is usually slower than *main memory*. However it is also much cheaper, and thus it is widely used to store large quantities of data that do not fit in the more expensive main memory. Usually secondary memory consists of media such as magnetic or optical discs or tapes. Data stored in secondary memory usually cannot be processed directly by the computer unless it is loaded into main memory. Main memory is also known as *RAM* (*random access memory*) and physically it is usually implemented by electronic integrated circuits.

same number is never generated twice for the same table. *Primary keys do not correspond to any of the attributes of an object*: they correspond to the object identity.

Figure e13.1 shows the class that is the basis for the following examples, and Table e13.1 shows the equivalent relational table with three instances of that class represented.

Although other formats could be used to specify the constraints on the columns here they are presented as acronyms for quick reference:

- *Uniq*, or *unique*, means that the column cannot repeat values.
- *Im*, or *immutable*, means that the value in the column cannot be updated.
- *NN*, or *not null*, means that the column does not admit the null value. It is exactly the opposite of the ≪*optional*≫ stereotype used for design classes. It is used here because *not null* is a common constraint implemented in databases.

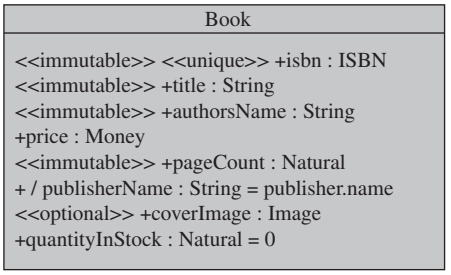

FIGURE e13.1

Reference class.

Table e13.1 Relational Table Equivalent to the Class of Figure e13.1

Table: Book							
PK,Uniq, Im,NN	Uniq,Im,NN	Im,NN	Im,NN	NN	Im,NN		NN
pkBook	isbn	title	authorsName	price	pageCount	cover Image	quantity InStock
10001	0553286587	Rama II	Arthur C. Clarke and Gentry Lee	6.99	466		2
10002	0553293370	Foundation and Empire	Isaac Asimov	5.99	282		3
10003	0671742515	The Long Dark Tea-Time of the Soul	Douglas N. Adams	6.99	307		21

- *PK*, or *primary key*, means that the column is the primary key of the table. If a column is *PK* it is necessarily immutable and not null. If only a *single column* is *PK* then it must be unique as well. However, if the *PK* is composite, spreading over more than one column, then each individual column may or may not be unique.

The first column in Table e13.1 is *pkBook*. Notice that it does not correspond to any attribute of the reference class. Its value is artificially generated by a *number sequence generator* so that it is unique for the whole application or at least for that table. Some authors (Wieringa & Jonge, 1991) refer to it as a *surrogate key*, that is, a value that has no semantic meaning, and is unique system-wide, never reused, system generated, and not handled by the user or application.

The other columns correspond to the attributes of the reference class:

- *isbn* is a unique attribute. Therefore the equivalent column is unique, immutable, and not null.
- *title*, *authorsName*, and *pageCount* are normal attributes that are immutable and mandatory. Therefore the equivalent column is immutable and not null, but not unique.
- *price* and *quantityInStock* are normal attributes that may change but cannot be null. Therefore the equivalent column is not null only.
- *coverImage* is an optional attribute that may be updated. Therefore, the respective column has no constraint: it may be updated and may be null.
- *publisherName* is a derived attribute. Therefore it is not represented in the relational table.

If an attribute of the design class is marked with ≪*transient*≫ then it also should not appear in the relational table.

13.2.1.1 Number sequence generator

Most database management systems provide number sequence generators that generate numbers that never repeat. This is necessary to provide values for primary keys, especially when multiple users are producing new records at the same time. The number sequence generator must assure that different users would not produce the same number at the same time.

Number sequence generators may be associated directly to the column of the table that contains primary keys. Every time a new record is inserted in the table, a new number in the sequence is created.

The database designer may choose an initial value for the sequence as well as an increment. For example, beginning with 500 with an increment of 5 the sequence generated would be 500, 505, 510, 515, etc.

13.2.1.2 Index selection

By default, a relational table is just a set of records. Finding a given object at a table would require iterating over all elements until the desired element is found. For example, looking for a book given its ISBN would require an exhaustive search.

Databases usually provide, however, the possibility of indexing columns. An *indexed column* has an auxiliary table that allows specific records to be found in almost constant time. For example, if the *isbn* column of the table is indexed, then when a book is searched based on its ISBN, no iteration is performed over the set of all records: the system simply would translate the value of the ISBN into a position in memory by using a hash function and retrieve the element from that

position. If that is not the desired element, then it looks for the next, and so on until finding it. If the hash function is well implemented and the hash table has enough space to avoid collisions (two values being translated to the same hash value), then usually the desired element is really in the first place searched, or very close to it at least.

The use of indices improves query speed. However, it slows database updating because every time a record is updated, inserted, or deleted, the auxiliary table must be updated as well (Choenni, Blanken, & Chang, 1993). Furthermore, indices also require more storage space for accommodating the auxiliary tables.

A primary key is indexed by default. Other columns may be indexed if the designer chooses to do that. Given the restrictions mentioned before, creating other indices may be an advantage in the case of an attribute that is used as internal qualifier. In that case, finding the objects quickly may be crucial for the application's performance. Otherwise, indices should be avoided. For example, columns that are rarely used for searching purposes (for example, a book's page count) should not be indexed.

13.2.2 Associations

Generally, associations between classes (except transient associations that do not persist) correspond to *associative tables* in the relational model, that is, tables with a primary key composed by the primary keys values of the tables that represent the participating classes.

In this case, the primary key of the associative table is in fact composed of two (or more) columns. Each column may repeat values individually depending on the association multiplicity; but the pair (or tuple) of values that compound the primary key can never be repeated.

Depending on the multiplicity of the association roles, some rules must be observed. Many-to-many associations will have no individual unique restrictions. However one-to-many and one-to-one associations require primary keys in which one or both columns are unique. This is explained in further details in the following subsections.

13.2.2.1 Many-to-many associations

If the association is many to many with both role multiplicities defined as *, then there is no restriction on the columns that compose the primary key of the associative table. Figure e13.2 shows an example of a many-to-many association.

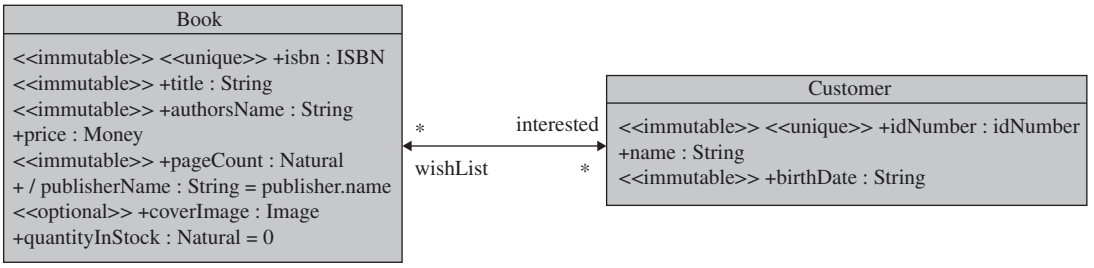

FIGURE e13.2

Many-to-many association.

Consider again the three books of Table e13.1 and the three customers shown in Table e13.2.

Notice that in Table e13.2 a customer has two codes: one is its *idNumber*, for example, a Social Security number. That value is known outside the information system. It is usually preformatted and although in normal conditions it does not change, nobody may prevent the government from changing all citizens' Social Security numbers if that is necessary. Therefore, though *idNumber* is unique for a customer, it does not qualify as a good primary key. The primary key for Table e13.2 is *pkCustomer*, which is a number created automatically by the number sequence generator; it is unique and assured never to change, because it has no meaning outside the database.

Table e13.3 shows an associative relational table that represents the links between some customers and some books desired by them. Notice that the PK spreads over two columns now. Each column contains a *foreign key* (*FK*), that is, a value that corresponds to the primary key of another table.

Table e13.2 Relational Table for the *Customer* Class

Table: Customer			
PK,Uniq,Im,NN	Uniq,Im,NN	NN	Im,NN
pkCustomer	idNumber	Name	birthDate
20001	987 – 65 – 4320	Abe	01/04/1970
20002	987 – 65 – 4329	Beth	02/23/1982
20003	987 – 65 – 4325	Charles	12/05/1979

Table e13.3 Associative Table Representing a Many-to-Many Association

Table: interested_wishList	
PK	
NN	NN
fkCustomer	fkBook
20001	10001
20002	10001
20001	10003

In the associative table *interested_wishList*, the columns *fkCustomer* and *fkBook* together form the *composed primary key*. Both of them cannot be null, and pairs of *fkCustomer*/*fkBook* cannot be repeated. However, as the association is many to many, each individual column may repeat values, as seen in the table.

Table e13.3 shows that Abe (customer 20001) desires the books "Rama II" (10001) and "The Long Dark Tea-Time of the Soul" (10003). Beth (customer 20002) only desires "Rama II" (10001), and Charles (20003) has no wishes.

Instead of naming the associative table with the names of the classes (*Customer_Book*), it is preferable to name it with the names of the roles (*interested_wishList*), because more than one association may exist between two classes.

If the association is mandatory in one direction or both directions, then special considerations must be observed:

- If the association is mandatory on *one* side, then all instances from the *other* side must appear at least once in the associative table. *If* in the example of Figure e13.2, the role *interested* was mandatory (1..*), then each book should have at least one associated customer. Then each primary key value from the *Book* table should appear in the *interested_wishList* table at least once.
- If the association is mandatory on *both* sides, then all instances from both classes should appear at least once in the associative table. If in the example of Figure e13.2, both roles were mandatory (1..*), then each book from the *Book* table and each customer from the *Customer* table should appear at least once in the associative table.

More generally, considering that *A* has an association to *B*, and that the lower bound of the *B* role is *n* while the upper bound is *m* (multiplicity is *n..m*), the number of times that each instance of *A* must appear in the associative table is at least *n*, and no more than *m*. For example, if the multiplicity of role *B* is 2..5, then each instance of *A* must appear in the associative table at least twice and no more than five times. Unfortunately that constraint is not usually present in database management systems.

13.2.2.2 One-to-many associations

When the association is one to many, then the column on the *many* side must have a *unique* constraint. This means that the column may not repeat elements individually while the other column may repeat elements. The elements that cannot be repeated in the associative table therefore can be linked to a single element of the other table; this constraint assures that the association is one to many. Figure e13.3 shows an example of a one-to-many association.

Table e13.4 shows an associative table for the one-to-many association represented in Figure e13.3. Notice that the *unique* constraint in the right column prevents the table from associating a book to more than one publisher.

As seen in Table e13.4, publisher 30001 has one book (10002) and publisher 30002 has two books (10001 and 10003). As books cannot be repeated in this table, no book can belong to more than one publisher.

It is also possible to represent associations from many to one as *foreign keys* in the table that represents the class at the *many* side of the association. For example, as each book may have only one publisher, then the association between book and publisher could be implemented as in Table e13.5.

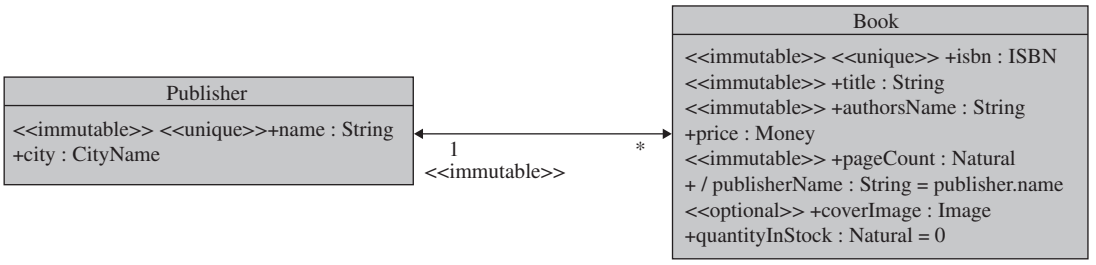

FIGURE e13.3

Example of a one-to-many association.

Table e13.4 Associative Table Representing a One-to-Many Association

Table: publisher_book	
PK	
Im,NN	Uniq,NN
fkPublisher	fkBook
30002	10001
30001	10002
30002	10003

The foreign key *publisher* is a direct reference to the primary key of the *Publisher* table. If the association is straight to 1, then the foreign key column must not be null. If the association is to 0..1, then the foreign key column should not have that constraint.

Although this approach is not as homogeneous as associative tables, it is usually preferred by designers because it avoids the need to implement a new table. One disadvantage is that in this case many-to-many associations are implemented as associative tables and many-to-one associations are implemented inside one of the original tables. Also, if the association is not mandatory (if its multiplicity is 0..1), and relatively few elements are associated, there would be lots of null values in the foreign key column, wasting storage space and degrading performance. However, if the association is straight to 1, this disadvantage does not apply.

Table e13.5 Alternative Way to Implement a Many-to-One Association Without an Associative Table

Table: Book								
PK, Uniq, Im,NN	Uniq,Im, NN	Im,NN	Im,NN	NN	Im,NN		NN	Im,NN
pkBook	isbn	title	authors Name	price	page Count	cover Image	quantity InStock	fkpublisher
10001	0553286587	Rama II	Arthur C. Clarke and Gentry Lee	6.99	466		2	30002
10002	0553293370	Foundation and Empire	Isaac Asimov	5.99	282		3	30001
10003	0671742515	The Long Dark Tea-Time of the Soul	Douglas N. Adams	6.99	307		21	30002

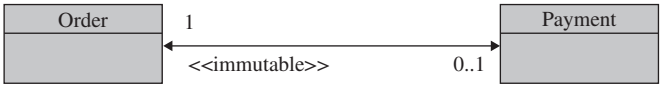

FIGURE e13.4

Example of one-to-one association.

13.2.2.3 One-to-one associations

One-to-one associations, mandatory or optional, require that the associative table have a *unique* constraint in both columns of the composed primary key in order to prevent any element on both sides from appearing more than once in the associative table. Figure e13.4 shows an example of one-to-one association that is mandatory on one side and optional on the other.

Table e13.6 shows the relational table that implements the one-to-one association of Figure e13.4.

As the role is mandatory for payments, all instances of *Payment* must appear in the associative table, but not all instances of *Order* must appear, because the role is not mandatory for them.

As in the case of many-to-one associations, one-to-one associations may also be implemented as foreign keys in one of the original tables. The foreign key column must necessarily be unique in that case. If the association role is also mandatory, then the foreign key column should not be null.

Table e13.6 Associative Table Representing a One-to-One Association

Table: order_payment	
PK	
Im,Uniq,NN	Uniq,NN
fkOrder	fkPayment
50001	60001
50003	60002
50005	60003
50011	60004
50016	60005
50021	60006
50030	60007

13.2.2.4 Ordered associations

An association with an ordered role (*sequence* or *ordered set*) may be implemented as an associative table with an extra column to represent the order of the element in the role's collection of elements. Figure e13.5 shows an example with two situations: ordered set and sequence (a list in which elements may be repeated).

The difference between the relational implementation of an ordered set and a sequence is that in the case of the ordered set the *order* column must not be included in the composed primary key, as shown in Table e13.7. However, in the case of a sequence (elements may be repeated), the *order* column must be part of the composed primary key, which in this case is composed of three columns (Table e13.8).

In Table e13.8, the fact that the *order* column is included in the primary key allows the same person to reserve the same book more than once (for example, 20001, Abe, has two reservations for book 10001, Rama II, in the first and fourth positions on the reservation list. A repetition like this would not be possible in Table e13.7, because the primary key does not include the *order* column: the same pair *fkBook*/*fkChapter* cannot appear more than once in the table, regardless of its position.

In both cases, to prevent a given position to be occupied more than once, a uniq constraint spreading over the origin class and the position should be defined. In the case of Table e13.7 and Table e13.8 the pair fkBook/order should be unique.

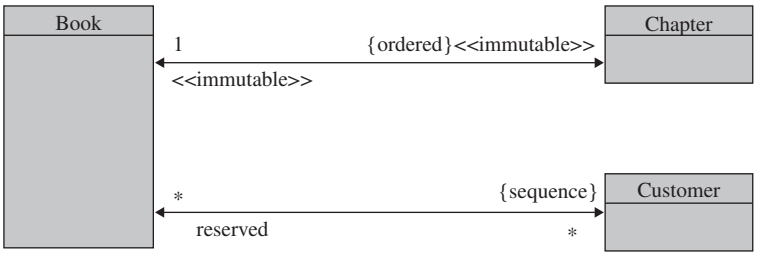

FIGURE e13.5

Example of ordered roles.

Table e13.7 Relational Table Representing an Ordered Set

Table: book_chapter		
PK		
Im,NN	Im,Uniq,NN	NN
fkBook	fkChapter	order
10001	130001	1
10001	130002	2
10001	130003	3
10002	130004	1
10002	130005	2
10003	130006	1
10003	130007	2

13.2.2.5 Associations representing bags

In the case of *bags*, in which elements may be repeated but have no position, the usual solution is to add an extra column to the associative table with a counter for the number of times a given pair participates in the association. Figure e13.6 shows an example of this kind of association.

Table e13.9 shows the implementation of the associative table for the example in Figure e13.6.

Table e13.8 Relational Table Representing a Sequence

Table: book_customer		
PK		
NN	NN	NN
fkBook	fkCustomer	order
10001	20001	1
10001	20003	2
10001	20002	3
10001	20001	4
10002	20003	1
10003	20001	1
10003	20002	2

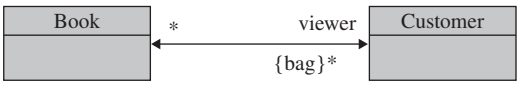

FIGURE e13.6

Example of a bag.

Table e13.9 specifies that Abe (20001) has viewed the book "Foundation and Empire" (10002) six times. Beth (20002) viewed "Rama II" (10001) twice, "Foundation and Empire" (10002) once, and "The Long Dark Tea-Time of the Soul" (10003) once. It is not necessary to represent in the table any pair whose quantity is zero; this is why Charles (20003) does not appear in the table: he has never viewed any book.

Table e13.9 Associative Table Representing a Bag

Table: book_viewer		
PK		
NN	NN	NN
fkBook	fkCustomer	quantity
10002	20002	1
10001	20002	2
10002	20001	6
10003	20002	1

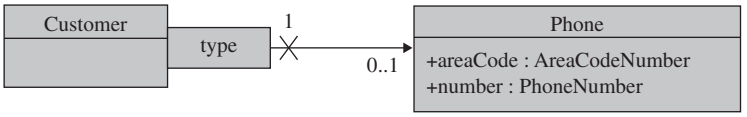

FIGURE e13.7

Example of a map with an external qualifier.

13.2.2.6 Qualified associations

In the case of a qualified association defined as a map (multiplicity 1 or 0..1) with an internal qualifier (the qualifier is an attribute of the qualified class), it is sufficient to implement the association as a regular one-to-many or many-to-many association depending on the multiplicity on the side of the qualifier, as explained in previous sections.

The only special care that the database designer must take in that case is to ensure that the column of the qualifier attribute is *unique* and *immutable*. It may be indexed if quick access to the records is necessary.

However, when the qualifier is *external*, it is necessary to add a third column to the associative table to allow for the representation of the qualifier. Figure e13.7 shows an example of that situation.

Table e13.10 shows the implementation for the map defined in Figure e13.7. The associative table has a primary key that is composed only of the origin class key and the qualifier. The destination class is left out of the primary key. However, it must be marked as *unique* because, in the example, each phone has a single type.

Table e13.10 Associative Table Representing a Map With an External Qualifier

Table: customer_phone		
PK		
NN	NN	Uniq,NN
fkCustomer	type	fkPhone
20001	Home	70001
20001	Cellphone	70002
20002	Home	70003

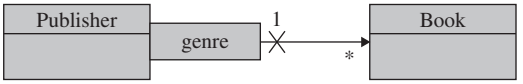

FIGURE e13.8

Example of a partition with an external qualifier.

If the external qualifier defines a partition (multiplicity *), as shown in Figure e13.8, it is implemented as shown in Table e13.10. But in the case of a partition with an external qualifier, the primary key must have three parts, including the origin and destination foreign keys as well as the qualifier. Also, as the origin role multiplicity is 1, the destination class column must be marked with *unique*. In the example shown in Table e13.11, this means that a book may not have more than one genre.

If the role multiplicity at the origin were * (defining a relation where a book could have more than one genre), the implementation would basically be the same. The only difference is that the *unique* constraint in the destination column (*fkBook*) should not exist.

13.2.2.7 Association classes

An association with an association class is represented in two parts: a relational table for the association class with its attributes, and an associative table for the association with a reference to the association class table.

Figure e13.9 shows an association class, and Tables e13.12 and e13.13 implement its relational equivalent.

Thus, one way to represent associations with association classes is to use a table to represent the association class (Table e13.12), and an associative table (Table e13.13) with three columns: the primary keys of the participating classes (which forms the composite primary key of the

Table e13.11 Associative Table Representing a Partition with an External Qualifier

Table: publisher_book		
PK		
NN	NN	Uniq,NN
fkPublisher	genre	fkBook
60001	sci-fi	10001
60001	sci-fi	10002
60002	humor	10003

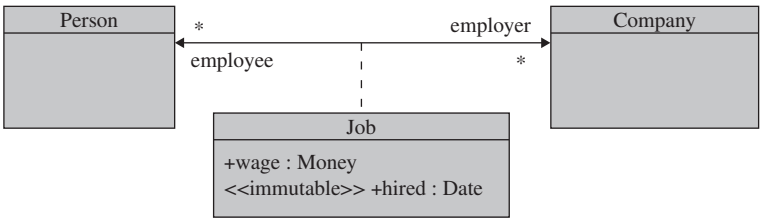

FIGURE e13.9

Example of an association class.

association) and the primary key of the association class, which is not part of the composite primary key of the association, but must be immutable and unique in that table.

A further constraint is that all values for *pkJob* in the *Job* table must appear in the *fkJob* column of the *employee_employer* table.

13.2.2.8 n*-ary associations*

In the case of associations among three or more classes (*n*-ary), an associative table is defined in which the primary key is formed by the primary keys of all participating classes. Figure e13.10 shows an example of a ternary association and Table e13.14 shows its relational representation.

In Table e13.14, pairs such as *fkBudgetItem/fkProject* may repeat elements (90001/100001, for instance). But the triple *fkBudgetItem/fkFinancialYear/fkProject* may never repeat.

Table e13.12 Relational Representation of an Association with an Association Class – Part 1: The Association Class

Table: Job		
PK,NN,Im,Uniq	NN	NN,Im
pkJob	Wage	Hired
80001	1,500.00	02/15/2008
80002	1,200.00	03/01/1999
80003	2,000.00	04/16/2005
80004	900.00	01/17/2001

Table e13.13 Relational Representation of an Association with Association Class – Part 2: The Association

Table: employee_employer		
PK		
NN	NN	NN,Im,Uniq
fkPerson	fkCompany	fkJob
20001	70001	80001
20001	70005	80002
20002	70001	80003
20003	70002	80004

13.2.2.9 Transient and façade controller associations

Transient associations are not represented in relational tables because, by their own definition, they exist only in primary memory, and it is not necessary neither desirable to persist them.

Some of the associations from the façade controller also do not need to persist. Associations from a controller to all instances of a class, which are mandatory for the instances, do not need to

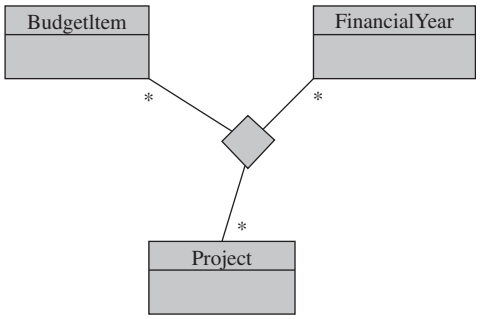

FIGURE e13.10

An example of a ternary association.

Table e13.14 Relational Equivalent of a Ternary Association

Table: budgetItem_financialYear_project		
PK		
NN	NN	NN
fkBudgetItem	fkFinancialYear	fkProject
90001	100001	110001
90001	100002	110002
90002	100001	110003

be transformed into association tables, because they always repeat the same value for the controller and have all the elements of the other side. That information is already available in the primary key column of the table representing the conceptual class, and, therefore, that association table would be redundant. For example, in Figure e13.11, the association between the controller and *Customer* with no explicit role name is mandatory for all customers. Thus, an associative table for representing that association would contain only the primary keys of all customers. As they are already represented in the *Customer* table, repeating them in a different table is unnecessary.

However, this observation is valid only if the association role is strictly 1 on the controller side. If the controller side has multiplicity 0..1, then the association should be represented separately. In Figure e13.11 the *premium* association must be represented because not every customer belongs to it.

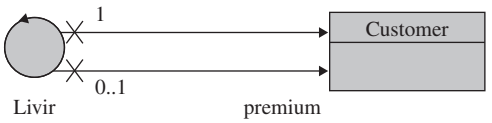

FIGURE e13.11

A mandatory and an optional association to a façade controller.

Table e13.15 Implementation of an Optional Association from the Façade Controller

Table: premium
PK
NN
fkCustomer
20001
20002

Not every customer is a *premium* customer,[4] and thus they must be listed somewhere. There are at least two choices for representing premium customers: adding a Boolean field to the *Customer* table indicating which customers are premium, or creating a single column table that contains the primary keys of premium customers.

The premium customer table does not have to include a column that represents the primary key of the controller (1) because the controller is not an entity (and therefore it has no primary key), and (2) because the controller is a singleton and even if a primary key is assigned to it, repeating the same value in all rows of the table would be unnecessary.

Table e13.15 shows a possible implementation for this optional association.

According to Table e13.15 only Abe (20001) and Bea (20002) are premium customers, and Charles (20003) is not.

13.2.3 Inheritance

Relational databases do not support inheritance directly. Mapping inheritance relations to relational databases is an issue that demands attention. There are many different approaches and this section discusses some of them. All examples in the following subsections are based on Figure e13.12.

[4]Note that this is a normal association and not a derived one.

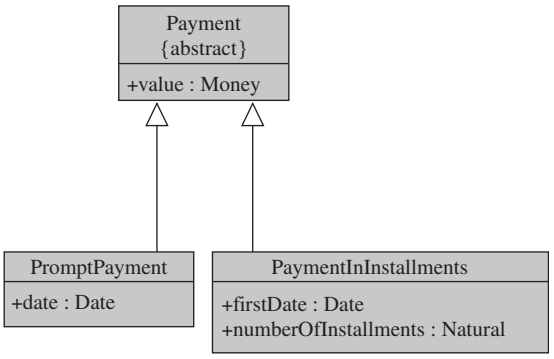

FIGURE e13.12

A situation with inheritance of attributes.

Table e13.16 Implementation of Inheritance in a Single Table with a *Type* Field

Table: Payment					
PK,NN,Uniq,Im	NN,Im	NN			
pkPayment	type	value	date	firstDate	numberOf Installments
200001	promptPayment	300.00	04/09/2014		
200002	promptPayment	251.00	07/02/2014		
200003	paymentIn Installments	1,890.00		12/09/2013	12

13.2.3.1 Implementing the entire hierarchy in a single table

One solution for representing inheritance that sounds straightforward at first glance is to implement the entire hierarchy in a single table. This table contains all the attributes of all classes in the hierarchy. It also has to identify which class is being represented in each record; this can be accomplished by adding a *type* column, as in Table e13.16.

Only attributes that belong to the class at the top of the hierarchy may not be null, because all other attributes could be null when the record belongs to one subclass and the attribute to another subclass. For example, if the type of the record is *promptPayment*, then *firstDate* and *numberOfInstallments* must necessarily be null. If it is assumed that an object cannot change its class after instantiation, then the *type type* should be immutable.

If multiple inheritance is used (for example, if a payment could be prompt and installments at the same time),[5] then the *type* column should be replaced by a set of Boolean columns: *isPromptPayment* and *isPaymentInInstallments*. In this case, a payment could be prompt, in installments, or both (or neither, if they could also be instances of *Payment*, which in the present example cannot occur because it is an abstract class).

This approach has some disadvantages. It is hard to manage consistency between subclasses because all data is stored in the same place. Managing it adequately would require various complex control mechanisms. Also, lots of fields in the table would be null all the time and the big table would be a very sparse one, especially if a big and complex hierarchy is being represented.

13.2.3.2 Each concrete class as a single table

Another approach to represent inheritance is to represent each concrete class as a separate table. Each table would contain the attributes of the concrete class and the attributes of all of its superclasses. Tables e13.17 and e13.18 show a possible implementation using that approach.

Table e13.17 Implementation of Inheritance Using a Table for Each Concrete Class: *PromptPayment*

Table: PromptPayment		
PK,NN,Uniq,Im	NN	NN
pkPromptPayment	value	date
200001	300.00	04/09/2014
200002	251.00	07/02/2014

Table e13.18 Implementation of Inheritance Using a Table for Each Concrete Class: *PaymentInInstallments*

Table: PaymentInInstallments			
PK,NN,Uniq,Im	NN	NN	NN
pkPayment	value	firstDate	numberOfInstallments
200003	1,890.00	12/09/2013	12

[5] Of course this cannot be true. It's just a supposition for the sake of the example.

If only attributes are inherited, this approach may work. However if superclasses define their own associations that must be inherited, then managing this becomes a headache because either an associative table should fill in data from different tables in a single column, or a single associative table should be implemented for each subclass that inherits the association.

13.2.3.3 Each class in a single table

A better choice for implementing inheritance given the aforementioned problems is to define one table for each class of the hierarchy, even the abstract ones. That way, attributes and association inheritance may be easily implemented. The main disadvantage is that for instantiating an object, a number of tables equal to the number of its superclasses plus one should be accessed. Tables e13.19 to e13.21 show how to implement this approach. There must be references from the subclasses' tables to each immediate superclass table.

Table e13.19 Implementation of Inheritance Using a Table for Each Class: *Payment*

Table: Payment	
PK,NN,Uniq,Im	NN
pkPayment	value
200001	300.00
200002	251.00
200003	1,890,00

Table e13.20 Implementation of Inheritance Using a Table for Each Class: *PromptPayment*

Table: PromptPayment		
PK,NN,Uniq,Im	NN,Uniq,Im	NN
pkPromptPayment	fkPayment	Date
300001	200001	04/09/2014
300002	200002	07/02/2014

Table e13.21 Implementation of Inheritance Using a Table for Each Class: *PaymentInInstallments*

Table: PaymentInInstallments			
PK,NN,Uniq,Im	NN,Uniq,Im	NN	NN
pkPaymentInInstallments	fkPayment	firstDate	numberOfInstallments
400001	200003	12/09/2013	12

13.3 Saving and loading objects

The equivalence between object-oriented design and the relational database is just part of the compatibility issue between these two models. It is necessary also to decide how and when objects will be loaded and saved to the database. Some designers prefer to determine themselves the moment when such operations should be performed. However, that *handcrafted* approach for saving and loading objects is subject to logic errors, and usually it pollutes the domain-level code.

In addition, if the designer is the one who decides when to save and load objects, sometimes those operations could be performed unnecessarily (for example, loading objects that are already in memory and saving objects that were not changed). Controlling these issues case by case, method by method is not the most productive way to develop software.

It is possible to implement the processes for saving and loading objects with automatic mechanisms. In this case, the designer should only decide which classes, attributes, and associations are persistent, and a whole set of methods and data structures will be automatically created to allow those elements to be loaded and saved at the appropriate moments. Initially this section presents the basic or naïve implementation of this mechanism. Later, the limitations of the technique are explained and possible solutions drafted.

13.3.1 Virtual proxy

In order to implement an automatic mechanism for saving and loading objects, we can use a design pattern called *virtual proxy* (Gamma, Helm, Johnson, & Vlissides, 1995). A virtual proxy is a very simple object that implements only two responsibilities:

- It must know the value of the primary key of the real object it represents.
- It must redirect to the real object all messages it receives in its name.

Below is a draft of the way a virtual proxy works:

```
Class VirtualProxy
  var realObjectPk:PrimaryKey
  for any message msg received do
    realObject:=BrokerManager.get(realObjectPk)
    realObject.msg()
  end for
end class
```

Later, in the following sections, the way the *BrokerManager* works is gradually explained.

Thus, the design with virtual proxies requires that instead of associating domain objects directly with other domain objects, they must be associated to their proxies. In this way, it is possible to bring into memory an instance of *Publisher* without loading all instances of *Book* that are linked to it. The instance of *Publisher* is associated to the *proxies of books*, which are very simple objects. The proxies are created in main memory and require much less space than the instances of the real class, such as those of *Book*. An instance of *Book* is loaded only if necessary, that is, only if a message is sent to it through its proxy. This economic way of using memory is called *lazy load*, and it is very efficient in terms of time and main memory in some situations.

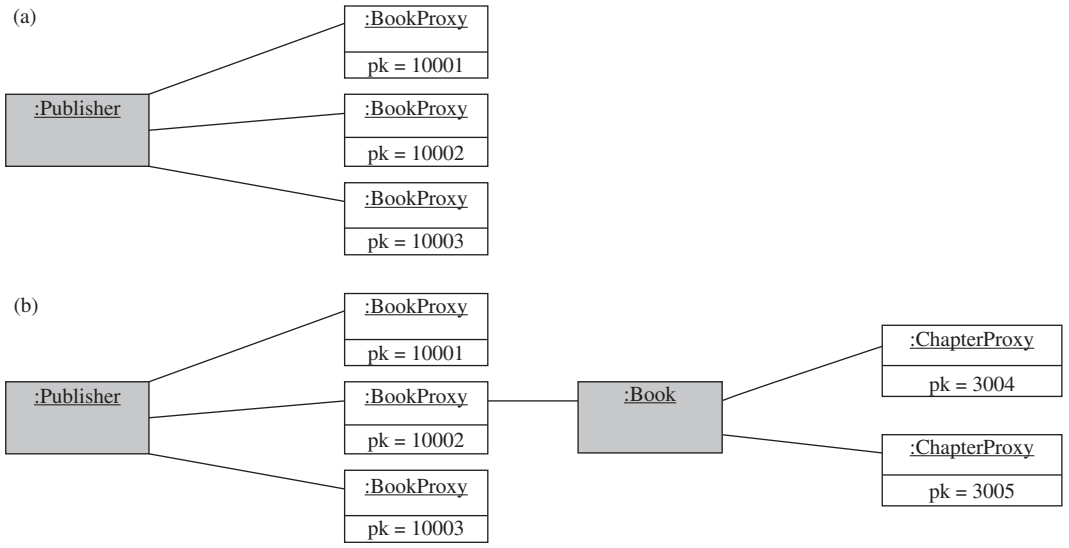

FIGURE e13.13

(a) Initial state in which only a publisher and book proxies are in main memory. (b) Situation after the publisher sends a message to one of the books.

To prevent the designer from worrying about when the objects must be loaded, the virtual proxy mechanism must be interposed to all persistent links in main memory. Real objects send messages to each other as if the proxies did not exist. But proxies intercept every message. The proxies ensure that the real object will be loaded if it is not in memory.

Figure e13.13 is an example of the lazy load mechanism. Initially (Figure e13.13a), only an instance of *Publisher* is in main memory. It is associated to three books. However, instead of having the books in memory, only their proxies are there. If the instance of *Publisher* must send a message to one of the books, it simply sends the message through the link; the message is intercepted by the proxy that calls the *BrokerManager*, which ensures that the book is loaded into the memory (Figure e13.13b). As the book is associated to some chapters, only the chapters' proxies are created in memory, not the real objects.

If one of the chapters receives a message from the book, then only that chapter would be brought into memory.

13.3.1.1 Virtual data structures

The implementation of virtual proxies for each object may be very inefficient when an object has many links; for example, a publisher with 50,000 registered books would demand the instantiation of 50,000 proxies to be associated to it when it is brought into memory. Fortunately, there is a way to avoid instantiating large quantities of proxies, which is the implementation of virtual data structures to physically replace the implementation of the associations in main memory.

Thus, a publisher would not have 50,000 links to 50,000 proxies of *Book*, but a single link to a *VirtualSet* structure with 50,000 PKs. The *VirtualSet* implements regular operations to add, remove, update, and query objects: the same operations a normal set must implement. The only difference is that it does not store the real objects, but only their primary keys. The virtual set does not bring objects into memory; it brings only their primary key numbers. The *VirtualSet* and its counterparts, *VirtualSequence*, *VirtualOrderedSet*, *VirtualBag*, *VirtualMap*, etc., use the *BrokerManager* to load real objects when necessary instead of holding all the real objects.

A virtual data structure may be implemented with the following principles:

- Instead of a physical representation of a collection of objects, it is a physical representation of a collection of the primary key values of the real objects.
- The method that adds an object to the collection must only add the primary key of the real object to the physical representation.
- The method that removes an object from the collection must only remove the primary key of the real object from the physical representation.
- Any method that performs a query on the data structure to return one or more objects receives the real object(s), which are requested from the broker manager.

Thus, adding and removing links between objects may be performed without having the real objects in memory (at least from one side of the link). An object is only brought into memory when information about it becomes necessary, that is, when it receives a message.

13.3.1.2 Discussion

Lazy load is useful because it only brings into memory objects that are going to receive a message. For example, if an instance of *Publisher* is in main memory and one of its books is going to be updated, then it is not necessary to load all instances of *Book* linked to the publisher, but only one.

On the other hand, if the instance of *Publisher* must search its books to find the most expensive of them, then all instances of *Book* that are linked to it must be loaded into main memory. This is the drawback of the technique. Performing all operations on objects in main memory can be an extraordinary waste of time. Imagine loading 50,000 instances of books to main memory just to discover which one of them has the highest price.

Virtual proxies work well when relatively few objects are brought into memory for each user operation. For example, a user searching for books to buy will only view a relatively small quantity of books in the list. That user would operate over just a single shopping cart and order. Those kinds of operations over small sets of objects are perfectly handled automatically by virtual proxies. However, when queries or commands involve iterating over large collections of objects, such as increasing the price of all books in the store, then using virtual proxies should be avoided, unless wasting processing time is not a problem; but that usually is not the case.

Thus, for performing queries or commands over large collections of objects, if the query or command uses just a few attributes of the objects, it would be advisable to consider replacing the virtual proxy mechanism in that specific case by a query or command directly performed over the database. Therefore, some messages, when they reach specific objects, would be redirected to a specific encapsulated implementation that performs the necessary actions in the database and returns the results as necessary, rather than being redirected to a proxy.

13.3.2 Brokers and materialization

The process of loading an object from the database into main memory is called *materialization*. Materialization is usually requested by a proxy from a *broker manager*, which may in turn delegate materialization to a *specialized broker*. The broker manager looks if the requested object is in memory. If it is not, then the broker manager activates the specialized broker to materialize the object.

Each class may have its own specialized broker. However, a single broker may also be implemented to serve all classes. A specialized broker must implement a method called *materialize* that does the following:

1. It creates in main memory an instance of the persistent class.
2. It initializes the values of the attributes of the new instance with values taken from the respective line and column in the database.
3. It loads and initializes the virtual data structures that implement the associations of the object with the primary keys of the respective linked objects.

In order to obtain the values for the primary keys of the linked objects, the specialized broker must know what associations are attached to the object being loaded; then it searches the occurrences of the primary key of the object in the associative tables that implement those associations. The primary keys of other objects associated to the primary key of the object being materialized are added into the virtual data structure, which has the responsibility of holding the respective association.

For example, *Broker4Book* should materialize instances of *Book*, as defined in Figure e13.14.

According to this conceptual model, *Broker4Book* must implement the *materialize* method by performing the following operations:

1. Create an instance of *Book*.
2. Fill in the *isbn*, *title*, *authorsName*, *price*, *pageCount*, *coverImage*, and *quantityInStock* attributes of the new instance with the values stored in the respective columns of the *Book* column in the database.
3. Search the *book_chapter* table for occurrences of the primary key of the book in the *fkBook* column. For every occurrence, add the corresponding value found in the *fkChapter* column to the virtual set *chapters* of the new instance of *Book*, which implements the association role for the book.

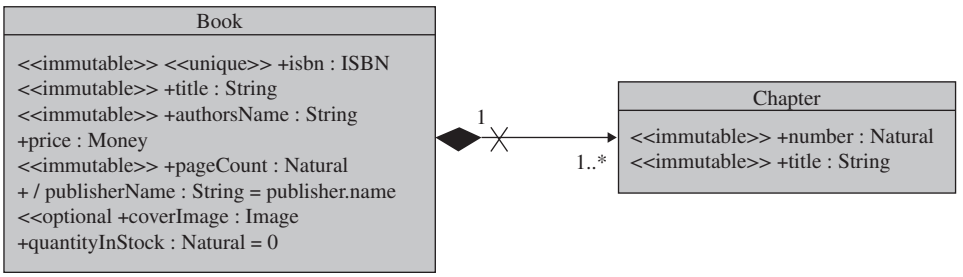

FIGURE e13.14

Reference conceptual model.

The materialization performed by the specialized broker must not be confused with the creation of a new instance as defined by the system operation contracts (Chapter 8). In the contracts, the creation of an instance refers to the insertion of new information in the system, independent of the physical storage (main memory or secondary memory). The materialization performed by the *broker* only refers to the operation of bringing into the main memory an existing object that is not there physically yet. Materialization is, therefore, an operation that belongs exclusively to the persistence tier; it has nothing to do with the business tier.

13.3.3 Caches

Objects in main memory may be classified as follows regarding their state related to the database:

- *Clean* or *dirty*, depending on whether or not they are consistent with the version stored in the database.
- *Old* or *new*, depending on whether or not they already exist in the database.
- *Deleted* or *kept*, depending on whether they have been deleted in main memory but not yet deleted in the database.

A *cache* is a data structure, similar to a *map* or *dictionary*, that associates primary key values to the real objects it represents.

Although there are eight possible combinations for the features defined above, in practice only four combinations are sufficient to manage objects in main memory:

- *Old clean cache*: Keeps objects that are consistent with the database.
- *Old dirty cache*: Keeps objects that exist in the database but have been updated in main memory and are, therefore, inconsistent with the database.
- *New cache*: Keeps objects that have been created in main memory, but which are not yet in the database.
- *Delete cache*: Keeps objects that were deleted in memory, but still exist in the database.

The broker manager verifies that an object is in main memory by performing a query on the existing caches, asking for the primary key of the object being requested. If the broker manager finds a reference to that object in one of the caches, it returns the reference to the proxy that asked for it. On the other hand, if the broker manager does not find any reference to the requested object in any cache, then it asks a specialized broker (as, for example, *Broker4Book*) to materialize the object. The object is then materialized and inserted into the *old clean cache*.

If, by any chance, that object is updated in main memory, that is, if some of its attributes or links have been changed, it must be moved to the *old dirty cache*.

The persistence mechanism must have ways to assure that every time the inner state of an object in the old clean cache is changed it is moved to the old dirty cache. This could be accomplished, for example, by adding a special command such as *BrokerManager.becomeDirty*(*self*) to any method that changes the object. If the object originally is in the old clean cache, that method will move it to the old dirty cache.

Objects that were created in memory as a result of a contract postcondition are stored into the *new cache*. An object that is in the new cache may not be moved to the old dirty cache, even if its attributes have been changed. It stays in the new cache until a *commit* is performed. After the commit it is moved to an old clean cache.

When an object is deleted from main memory, the result will depend on which cache the object came from. If it was in the old clean cache or in the old dirty cache, then it is moved to the *delete cache*. However, if it was in the new cache, then it may be simply deleted from main memory.

13.3.3.1 Commit and rollback

The *commit* and *rollback* operations are usually activated by the interface tier to indicate that a transaction was successful and confirmed, or that it was cancelled, respectively. These operations are implemented by the broker manager. In the case of a commit, the broker manager should do the following:

1. Perform an *update* in the database for every object in the old dirty cache, and move those objects to the old clean cache.
2. Perform an *insert* in the database for every object in the new cache, and move those objects to the old clean cache.
3. Perform a *remove* in the database for every object in the delete cache, and delete those objects from main memory.

In the case of a *rollback*, the broker manager must simply remove all objects from all caches, except those in the old clean cache.

As the old clean cache may grow indefinitely, it is necessary to implement some mechanism to remove the oldest objects from it every time its size reaches some previously established threshold.

The other caches only grow up to the moment a commit or rollback is performed. At that moment, they are emptied.

13.3.3.2 Cache control in a multiuser server

If more than one user connects to the system, it is necessary to determine how to share objects among different users. Assuming we have a client/server architecture with interface, domain, and persistence tiers, at least two approaches are possible:

- *The three tiers are executed in the client*. There is no main memory share in the server, only the database. The client is heavy and the server is used only to access or store data after a commit is performed. In this case, what travels in the network is data in the form of relational table records and SQL[6] instructions. Information travels only when an object must be materialized or committed. The disadvantage of this design is that the client node is overloaded. However, client applications have recently become richer and more complex: not depending on a server to process the logic of the application may be a crucial requirement.
- *The domain and persistence tiers are implemented in the server, and the interface tier is implemented in the client*. In this case, the objects will exist in main memory *only in the server*, and what travels through the network are the parameters of the system operations and the returns from queries. The advantage is that clients are lighter, and usually it is cheaper to upgrade a server than upgrading thousands of clients. However, for some applications that require fast processing of the data, waiting for the server may not be viable.

[6] *Structured Query Language*, the dominant language for defining, accessing, and updating relational databases (Date, 1982).

If the objects are physically in the server only, there are still more possibilities. In one case, all users share the four caches, with the disadvantage that a user could have access to objects that are being modified but have not yet been committed by another user. This option seems to be unadvisable for most applications.

The other option is to share only the objects in the old clean cache among the users. There should be multiple instances of other caches that are private to each user. If an object is in a private cache of one user, then the other users cannot access it. They must wait for a commit or rollback from the first user. Thus, the persistence mechanism for multiuser access could be implemented like this:

- An old clean cache is shared by all users.
- Each user has a private old dirty cache, delete cache, and new cache.

By doing this, it is possible to ensure that no user would have access to objects that are being modified by other users. Therefore, it is possible to use the caches to implement a *lock* mechanism, that is, when a user is updating an object, the other users cannot access it. Only when the user that has the object performs a commit or rollback, and the object is moved to the old clean cache or cleaned from main memory, can the other users gain access to it again.

An advantage of this method is the optimized usage of server main memory. All users share the objects in the old clean cache, which is the only one that grows indefinitely. The other four caches, which are specific for each user, only grow during a transaction. When a commit or rollback happens, those caches are emptied.

However, this may negatively affect scalability, because some users could be stuck while others are dealing with some objects. It is possible to implement a more sophisticated control mechanism based on optimistic merges: two or more users may edit the same object as long as they do not update the same attribute or association.

This must be used with care, however. Concurrency issues are always complicated concerns and many decisions may depend on business rules. For example, may the price of a book be updated while a user is finishing an order?

Some applications could even require a more pessimistic locking strategy: if a user is browsing an object, no other user may even have access to view it. This is the case for banking applications, for example, which require maximum data security.

13.4 The whole process

	Inception	Elaboration
Business Modeling	Build a general view of the system: • Build a business use case diagram and determine the automation scope for the project. • Build preliminary activity diagrams for business use cases. • Build preliminary state machine diagrams for key business objects.	

(*Continued*)

(Continued)

	Inception	**Elaboration**
Requirements	Prepare the system use case diagram (functional requirements): • Identify the system actors from the business use case model. • Identify the system use cases from the business use case model, and the activity and state machine diagrams from business modeling. Identify nonfunctional requirements as use case annotations: • Identify the main business rules associated to use cases. • Identify the main quality issues associated to use cases. Identify the supplementary requirements.	Detail requirements by expanding use cases: • Identify the main flow. • Identify the alternate flows: variants and exception handlers.
Analysis and Design	Prepare the preliminary conceptual model by observing system use cases and the concepts needed by them.	Elaborate the system sequence diagrams: • Represent the main flow of a use case as a system sequence diagram. • Represent the system commands and queries using stateful or stateless strategies. • Complete system sequence diagrams with alternate flows. Refine the conceptual model: • Identify concepts, attributes, and associations in the text of expanded use cases. • Detail attributes and associations with stereotypes, multiplicity, and constraints as needed. • Organize the model by using inheritance, association classes, and temporal specifications. • Add invariants as needed. • Improve the conceptual model with the application of analysis patterns. Write system operation contracts for commands and queries in system sequence diagrams: • Identify preconditions and exceptions based on invalid parameters and complementary constraints. • Identify postconditions for commands and returns for queries.

(*Continued*)

(Continued)

	Inception	Elaboration
		Design the domain tier: • Create a sequence or communication diagram for each system command contract. • Use those diagrams to decide which methods to implement in each class. • Inspect those diagrams to discover which associations can be unidirectional and define them as such. • Look at system query contracts and decide which delegate queries could be implemented; some of them could be derived attributes or derived associations. • Produce or refine the design class diagram. Design the interface tier: • Design the top-level interface organization based on areas and pages. • Refine pages, including view components for parameters of system operations and results for system queries. • Associate operation activation to interface events such as buttons and menus. Design the persistence mechanism: • **Reuse or build a persistence mechanism based on ORM, virtual proxies, caches, and brokers.** • **Decide where to apply the default persistence mechanism and where to implement direct database access for performance optimization.**
Implementation		Generate code: • Generate code for classes. • Generate code for attributes. • Generate code for associations. • Generate code for necessary basic methods such as get, set, add, remove, create, destroy, and others. • Generate code for system operations and delegate methods using dynamic models as reference.

(*Continued*)

(Continued)

	Inception	Elaboration
Test		Plan and execute tests: • Perform unit tests for methods and classes that were not generated automatically. • Perform system operation tests based on system operation contracts. • Perform use case tests based on use case scenarios.
Project Management	Estimate total effort, ideal calendar time, and average team size for the project. Estimate the duration and quantity of iterations for each phase. Prepare the phase plan and iteration plan for the first iteration.	

13.5 Questions

1. What kind of associative table should be used to represent an association that is ordered in both directions? How do you represent it in the case of two ordered sets, or two sequences, or one ordered set and one sequence?
2. What kind of association table should be used to represent an association with an association class that is marked with {*bag*} on one side?