CHAPTER

Code Generation e10

KEY TOPICS IN THIS CHAPTER:

- Code generation for classes, attributes, and associations
- Code generation for delegated methods and system operations
- Patterns for filtered queries

10.1 Introduction to code generation

The *Implementation* discipline of UP includes code generation activities for design models. It is necessary to generate code for the classes from the domain tier, and also for the other technological tiers of the system. This chapter concentrates on code generation for the domain tier.

Once the communication diagram and the DCD are produced, code generation is an activity that may be systematized to a point that it can be practically done automatically. Automatic code generation is feasible for domain classes, which perform all the logical processing specified by the system operation contracts.

This chapter presents rules for code generation for the DCD and communication diagrams. Examples are presented in pseudocode, which may be translated to most programming languages (preferably object-oriented ones).

10.2 Classes and attributes

DCD classes are usually directly converted into programming language classes. Class attributes are converted into *private instance variables* in the respective class.

As explained in Chapter 6, attribute types should be alphanumeric (such as *Integer*, *Real*, *String*, *Boolean*, etc.), primitive (such as *Date*, *Money*, *Isbn*, etc.), or enumerations (such as *CalendarDay*, *Gender*, *PhoneType*, etc.).

If other objects can access the attribute then it should be implemented with a getter method. If the attribute can be updated then it must be implemented with a setter, which can be a straight *setAttribute* method or another type such as *incrementAttribute* if the attribute is numeric, for example. Figure e10.1 presents an example of a design class that is used to show how programming code is generated.

The following code corresponds to the pseudocode implementation of the class shown in Figure e10.1. First, the attributes of the class are transformed into programming code private attributes:

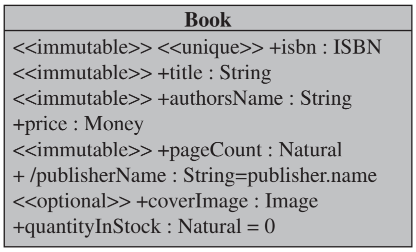
Book
<<immutable>> <<unique>> +isbn : ISBN
<<immutable>> +title : String
<<immutable>> +authorsName : String
+price : Money
<<immutable>> +pageCount : Natural
+ /publisherName : String=publisher.name
<<optional>> +coverImage : Image
+quantityInStock : Natural = 0

FIGURE e10.1

A reference class for code generation.

```
CLASS Book
  PRIVATE ATTRIBUTE VAR isbn:Isbn
  PRIVATE ATTRIBUTE VAR title:String
  PRIVATE ATTRIBUTE VAR authorsName:String
  PRIVATE ATTRIBUTE VAR price:Money
  PRIVATE ATTRIBUTE VAR pageCount:Natural
  PRIVATE ATTRIBUTE VAR coverImage:Image
  PRIVATE ATTRIBUTE VAR quantityInStock:Natural
  ...
```

Observe that all attributes, except for *publisherName*, which is derived, are defined as pseudo-code attributes. Most languages would not differentiate *ATTRIBUTE VAR* from *ASSOCIATION VAR*. In this book those names are used to make a clear distinction between variables that represent attributes and variables that implement association roles. However, for most languages any instance variable would be simply declared as *VAR*, or an equivalent language specific expression.

On the other hand, most languages differentiate between PRIVATE and PUBLIC variables and methods. Whenever private variables are allowed, they are the best choice, because this way, attributes are encapsulated into the object, and they may not be changed by objects that have no authorization to do that.

Now, we must consider how the class attributes are going to be updated. Some of them can be updated virtually at any time, while others should be defined only at creation time and prevented from changing after that: these are *immutable attributes*. In Figure e10.1 attributes that must not be changed after the object is created are stereotyped as ≪*immutable*≫.

A set of *setters* and one[1]*constructor* may be defined following these recommendations:

- The constructor must receive as parameters the initial values for all attributes that are not optional, derived or with a defined initial value (if any).
- The implementation of the constructor initializes attributes with initial values with the respective values (if any).
- The constructor must receive as parameters the objects to fill the mandatory roles (if any) of the associations (later this will be shown in a more complete example).

[1]In some cases, classes may allow more than one constructor, though.

• Only attributes that are not derived and immutable should define one or more setters (*setAttribute*, *incrementAttribute*, etc., depending on the case).

Thus, the following code may be created as a continuation of the definition of the implementation class that started above:

```
...
CONSTRUCTOR METHOD create(anIsbn:Isbn; aTitle:String,
  anAuthorsName:String; aPrice:Money; aPageCount:Natural)
  isbn: = anIsbn
  title: = aTitle
  authorsName: = anAuthorsName
  price: = aPrice
  pageCount: = aPageCount
  quantityInStock: = 0
END CONSTRUCTOR METHOD
METHOD setPrice(aPrice:Money)
  price: = aPrice
END METHOD
METHOD raisePrice(aPercentage:Percent)
  price: = aPrice*(1 + aPercentage)
END METHOD
METHOD setCoverImage(aCoverImage:Image)
  coverImage: = aCoverImage
END METHOD
METHOD increaseQuantityInStock(anIncrement:Integer)
  quantityInStock: = quantityInStock + anIncrement
END METHOD
...
```

Notice that inside the constructor, each attribute is assigned to the respective argument and that *quantityInStock* is assigned 0 (its initial default value).

Only three attributes may be updated. The attribute *price* has two setters: *setPrice*, which defines a brand new price independent from the original one, and *raisePrice*, which raises the price by a percentage.

The *quantityInStock* attribute is updated only by adding an increment to it. As the *anIncrement* parameter may be negative, decrements may be done with this method as well.

Let us suppose now that the design produced interaction diagrams that demonstrate that all attributes of this class must be accessed by other objects. In this case, all attributes must implement a *getter* method, even the derived attribute. Thus, the following code may be a continuation of the code started above:

```
...
METHOD getIsbn():Isbn
  RETURN isbn
```

```
END METHOD

METHOD getTitle():String
  RETURN title
END METHOD

METHOD getAuthorsName():String
  RETURN authorsName
END METHOD

METHOD getPrice():Money
  RETURN price
END METHOD

METHOD getPageCount():Natural
  RETURN pageCount
END METHOD

METHOD getPublisherName():String
  RETURN ...²
END METHOD

METHOD getCoverImage():Image
  RETURN coverImage
END METHOD

METHOD getQuantityInStock():Natural
  RETURN quantityInStock
END METHOD
```

At this point the reader could be asking why *authorsName* is still an attribute typed as *String* and not a class with a many-to-many association to *Book*. The reason is that the example is simulating an ongoing project and not the final version of it. Up to this point, the only use cases examined did not require authors to be addressed as a complex concept. The *authorsName* attribute is perfectly adequate for the functionalities discovered to this point. We may even believe that this would change as other use cases are examined in the future, but as William of Ockham (1495) said "*Numquam ponenda est pluralitas sine necessitate*" (plurality must never be posited without necessity). This also adheres to the agile principle of never implementing functionality before it is necessary just because it would be easy to do so now.

10.3 Unidirectional associations

Unidirectional associations with no multiplicity constraints at the origin and unidirectional associations from singletons such as the façade controller may be implemented similarly to attributes as *instance variables*, and if necessary they should have methods for updating and querying.

[2]This method involves associations, which are explained in Sections 10.3 and 10.4.

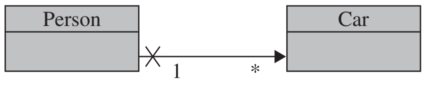

FIGURE e10.2

Unidirectional association with multiplicity constraint at the origin role.

However, if there are multiplicity constraints at the origin role, these associations must be handled as *bidirectional*, even if they are navigable only in one direction. Figure e10.2 shows an example of that situation, where although the association is only navigable from *Car* to *Person*, every car must be linked to a single person. If the association is implemented as an instance variable in the *Person* class, it will be hard to assure that a car belongs to a single person.

There are still some considerations we must discuss about the differences between an attribute and a unidirectional association. First, attributes are always implemented by variables whose types are alphanumeric, primitive, or enumerations. Associations, on the other hand, are implemented by variables whose types are domain classes (in the case of associations to one) or data structures (in the case of associations *to many*).

In addition, considering different role multiplicities and other features of associations, there are distinctions to be made regarding the methods to be implemented for each type of association.

Code generated for transient or persistent associations and attributes is the same. The only difference between these kinds of elements resides in the way they are stored.

In general and if necessary, a class may implement three kinds of methods for each association:

- Methods to add links: Usually referred to as *addRole*.
- Methods to remove links: Usually referred to as *removeRole*.
- Methods to get linked objects. Usually *getRole* would be implemented to return the whole set of linked objects. This set *must be protected* in the sense that it may be consulted and its elements may be iterated, but no element can be added or removed from the original set. Additionally, methods to get a specific element or subset given a key or other search criterion may be implemented if necessary.

Usually associations to one may be implemented as a single variable and not as a collection. In this case, the *addRole* and *removeRole* methods do not apply and the class implements instead a *replaceRole* method that replaces the current link with a new link.

Associations to 0..1 also require the implementation of the *replaceRole* method. But, in this case, *addRole* and *removeRole* may be implemented as well. If the *addRole* and *removeRole* methods are implemented for 0..1 roles, then they must check the role bounds.

Derived associations must implement only the *get* method, in accordance with their definition.

Other methods would still be necessary depending on the kind of the association, such as:

- If the association is *qualified*, there may be an additional *get* that receives as an argument the qualifier key and returns the qualified object or subset. Additionally, if the qualifier is external, the *add* method should receive the value for the qualifier; if it is an internal qualifier that argument must not be passed because it may be obtained from the object itself. A new *remove* method also may be added to remove a link to an object based in the value of its qualifier.

- In the case of *ordered* associations, an additional *get* method may return an object based on its position in the collection. Also, the *add* method could add elements at a given position, which is indicated as a new parameter for the method, and the *remove* method may remove a link from a given position. Ordered associations may also have methods to access, add, and remove elements from their head or tail.
- *Stacks* and *queues* may have special methods such as *push* and *pop* that follow specific rules for those structures.

Table e10.1 presents a summary of the methods that *may* be implemented for each kind of association in a class, depending on the design needs.

Observe that in Table e10.1, each kind of association has a different set of methods for accessing and changing links. The implementation of these methods follows definitions that are usually kept the same from class to class. The following subsections show examples of some of those methods.

10.3.1 Unidirectional association to one

The unidirectional association with role multiplicity 1 may be stored in a single instance variable in the origin class, and its type should be the destination class. Figure e10.3 shows a unidirectional association *to one* from *Car* to *Person* with role name *owner*.

The implementation of the *Car* class requires an instance variable named *owner* implemented with type *Person*. Regarding the association methods, following Table e10.1, only *getOwner* and *replaceOwner* should be implemented. The pseudocode for the *Car* class in this case could be

```
CLASS Car
  PRIVATE ASSOCIATION VAR owner:Person
  CONSTRUCTOR METHOD Create(anOwner:Person)
    owner: = anOnwer
  END CONSTRUCTOR METHOD

  METHOD getOwner():Person
    RETURN owner
  END METHOD

  METHOD replaceOwner(newOwner:Person)
    owner: = newOwner
  END METHOD

END CLASS
```

In the case of a role with multiplicity 0..1, there are two possibilities:

- It may be implemented as a set, that is, similar to the role with multiplicity *. In this case, if there is no object linked, the *get* method would return the empty set.
- It may be implemented as a single variable, that is, similar to the roles with multiplicity 1. In this case, as shown above, if there is no object linked, the *get* method would return the *null* object.

The second approach is widely adopted. There is even the possibility of using the *null object* design pattern (Woolf, 1998) instead of the language-provided null value. The advantage is that the

Table e10.1 Typical Operations over Associations Depending on their Type

Type	**To 1**	**To 0..1**	**To Many (*)**
Get	*getRole():Object*	*getRole():Object*	*getRole():Set*
Add	Does not apply	*addRole(obj)*	*addRole(obj)*
Remove	Does not apply	*removeRole()*	*removeRole(obj)*
Replace	*replaceRole(obj)*	*replaceRole(obj)*	*replaceRole(oldObj,newObj)*
Type	**Ordered Set**		**Sequence**
Get	*getRole():OrderedSet*		*getRole():Sequence*
	getRole(position):Object		*getRole(position):Object*
	getFirstRole():Object		*getFirstRole():Object*
	getLastRole():Object		*getLastRole():Object*
Add	*addRole(position,obj)*		*addRole(position,obj)*
	addFirstRole(obj)		*addFirstRole(obj)*
	addLastRole(obj)		*addLastRole(obj)*
Remove	*removeRole(obj)*		
	removeRole(position)		*removeRole(position)*
	removeFirstRole()		*removeFirstRole()*
	removeLastRole()		*removeLastRole()*
Replace	*replaceRole(oldObj,newObj)*		*replaceRole(position,obj)*
	replaceRole(position,obj)		*replaceFirstRole(obj)*
	replaceFirstRole(obj)		*replaceLastRole(obj)*
	replaceLastRole(obj)		
Type	**Map (Internal Qualifier)**		**Map (External Qualifier)**
Get	*getRole():Set*		*getRole():Set*
	getRole(key):Object		*getRole(key):Object*
Add	*addRole(obj)*		*addRole(key,obj)*
Remove	*removeRole(obj)*		*removeRole(obj)*
	removeRole(key)		*removeRole(key)*
Replace			*replaceRole(oldKey, newObj)*
	replaceRole(oldObj,newObj)		*replaceRole(oldObj, newObj)*
Type	**Partition (Internal Qualifier)**		**Partition (External Qualifier)**
Get	*getRole():Set*		*getRole():Set*
	getRole(key):Set		*getRole(key):Set*
Add	*addRole(obj)*		*addRole(key,obj)*
Remove	*removeRole(obj)*		*removeRole(obj)*
Replace	*replaceRole(oldObj, newObj)*		*replaceRole(oldObj,newObj)*

(*Continued*)

Table e10.1 (Continued)

Type	Set with Association Class	Bag	Array (Fixed Size)
Get	*getRole():Set* *getAssociationClass():Set* *getAssociationClass(obj):Object*	*getRole():Bag*	*getRole():Array* *getRole(position):Object*
Add	*addRole(obj)*	*addRole(obj)*	Does not apply
Remove	*removeRole(obj)* *removeAssociationClass(obj)*	*removeRole(obj)*	Does not apply
Replace	*replaceRole(oldObj,newObj)*	Does not apply	*replaceRole(position,obj)*

Type	Stack	Queue
Get	getRole():Object	getRole():Object
Add	pushRole(obj)	queueRole(obj)
Remove	popRole()	removeRole()
Replace	Usually does not apply	Usually does not apply

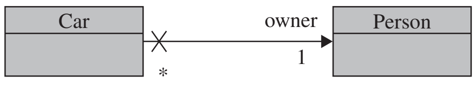

FIGURE e10.3

A class with unidirectional association to 1.

null object is language independent. The null object is an instance of a *Null* class whose behavior consists of doing nothing. If there is iteration over a *null object* it produces nothing, just as with an empty set. On the other hand, iterating over a null or nil *value* would produce an exception, which would not be desired in that case.

If we assume that the role multiplicity in Figure e10.3 is 0..1, there are two possible implementations. The first one considers the association as a set:

```
CLASS Car
  PRIVATE ASSOCIATION VAR owner:Set<Person>
  CONSTRUCTOR METHOD Create()
    owner:=Set.new()
  END CONSTRUCTOR METHOD

  METHOD getOwner():Set<Person>
    RETURN owner.protected()
  END METHOD

  METHOD addOwner(newOwner:Person)
    IF owner.size()>0 THEN
```

```
      Exception.throw('Car already has an owner')
    ENDIF
    owner.add(newOwner)
  END METHOD
  METHOD removeOwner()
    IF owner.size() = 0 THEN
      Exception.throw('Car does not have an owner')
    ENDIF
    owner.removeOneElement()
  END METHOD
  METHOD replaceOwner(newOwner:Person)
    IF owner.size() = 1 THEN
      owner.removeOneElement()
    ENDIF
    owner.add(newOwner)
  END METHOD
END CLASS
```

Every time a method such as *getOwner* returns a set of objects, the collection must be protected against change. That is explained in more detail later in this chapter.

The *removeOwner* method was implemented by calling the *removeOneElement* method over a set. That method removes one element (any one) from the set without the need to specify which one. As the set would have only one element at this point it would not be necessary to know which element it is in order to remove it from the set. The *removeOwner* method could also be implemented as *owner*: = *Set.new*(), with the same final result. However, as most programmers should notice, that implementation would produce a lot of garbage in memory if objects are often added and removed from that role, and this could degrade performance.

The second approach considers the association as a single variable, and uses the Null object design pattern:

```
CLASS Car
  PRIVATE ASSOCIATION VAR owner:Person
  CONSTRUCTOR METHOD Create()
    owner:=NullObject.instance()
  END CONSTRUCTOR METHOD
  METHOD getOwner():Person
    RETURN owner
  END METHOD
  METHOD addOwner(newOwner:Person)
    IF owner < > NullObject.instance() THEN
      Exception.throw('Car already has an owner')
    ENDIF
    owner: = newOwner
  END METHOD
```

```
    METHOD removeOwner()
      IF owner = NullObject.instance() THEN
        Exception.raise('Car does not have an owner')
      ENDIF
      owner: = NullObject.instance()
    END METHOD
    METHOD replaceOwner(newOwner:Person)
      owner: = newOwner
    END METHOD
  END CLASS
```

Both implementations of *replaceOwner* shown above assume that if the owner exists it is replaced, and that if it does not exist it is defined. No exception is raised if *replaceOwner* is called for a car that does not have an owner. In that case, it behaves just like *addOwner*.

10.3.2 Unidirectional association to many

The unidirectional association *to many* may be implemented as a data structure. If it is a simple association with multiplicity * in the destination role, then it may be implemented as a set. Below is a pseudocode example for the association represented in Figure e10.4:

```
CLASS Customer
  PRIVATE ASSOCIATION VAR wishes:SET<Book>
  METHOD getWishes():SET<Book>
    RETURN wishes.protected()
  END METHOD

  METHOD addWish(aBook:Book)
    wishes.add(aBook)
  END METHOD

  METHOD removeWish(aBook:Book)
    wishes.remove(aBook)
  END METHOD
END CLASS
```

It is not necessary to implement the *replaceWish* method here, because wishes usually are only added and removed from a customer. It is not common to replace a wish with another. Thus, that method is not considered for implementation.

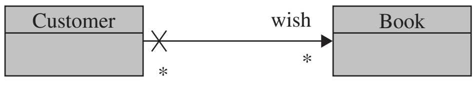

FIGURE e10.4

A class with a simple unidirectional association to many.

The *protected* message sent to a collection produces a noneditable version of that collection, to avoid elements being removed or added to the original collection by means other than the *add* and *remove* messages that are implemented in the class.

The implementation of that protection may be done in a number of ways; some of them are language specific. Examples include making a copy of the original collection (shallow copy or deep copy depending on the degree of protection intended), or using the *protection proxy* design pattern (Gamma, Helm, Johnson, & Vlissides, 1995), which suggests the implementation of a class that encapsulates the original collection, and implements only a method to iterate over the elements but not to modify the original collection.

If the association role is labeled with {*ordered*} or {*sequence*} the data type of the variable that represents the role must be replaced by the corresponding data type supported by the language, and additionally, other specific methods must be implemented, as mentioned in Table e10.1.

In the case of a role with identical lower and upper bounds, it may be implemented as an array. For example a multiplicity of 5 could be implemented as an array of 5 positions. The original *add* and *remove* commands are not applicable to the array structure because its size cannot be changed. However, elements may be replaced based on their position, as shown in Table e10.1.

Roles whose multiplicity is an interval, such as 3..8, may be implemented just like roles to many (*). The only difference is that the collection must have its bounds checked when objects are added or removed from it.

10.3.3 Unidirectional qualified association

The unidirectional qualified association is implemented in a manner similar to the association with multiplicity to many. However, instead of the data type *Set*, we use a mapping, or dictionary structure (*MAP*) that associates an alphanumeric, primitive, or enumeration type (key) to one object or a set of objects (values).

As for any other unidirectional associations, we must keep in mind that the unidirectional implementation is only possible when the association has no multiplicity bounds in its origin, that is, the origin role must have multiplicity *. If this is not the case, a bidirectional implementation must be considered.

Figure e10.5 shows the implementation of a qualified association defining a map to 0..1 with an internal qualifier. The customer wish list is now considered a qualified association. The respective code is presented below:

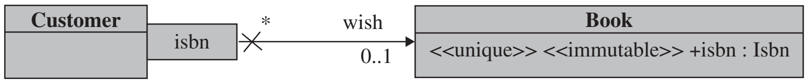

FIGURE e10.5

Qualified association with an internal qualifier.

```
CLASS Customer
  PRIVATE ASSOCIATION VAR
      wishes:MAP<Isbn,Book>
  METHOD getWishes():SET<Book>
    RETURN wishes.getValues()
  END METHOD
  METHOD getWish(anIsbn:Isbn):Book
    RETURN wishes.atKey(anIsbn)
  END METHOD
  METHOD addWish(aBook:Book)
    wishes.add(aBook.getIsbn(),aBook)
  END METHOD
  METHOD removeWish(aBook:Book)
    wishes.removeValue(aBook)
  END METHOD
  METHOD removeWish(anIsbn:Isbn)
    wishes.removeKey(anIsbn)
  END METHOD
  METHOD replaceWish(oldBook,newBook:Book)
    -- not implemented
  END METHOD
END CLASS
```

The *replace* method is not implemented for wishes again because it would not make sense to replace a book with another.

We note here that the basic *map* data structure has the usual operations for accessing a value given its key (*atKey*), accessing the set of all values (*getValues*), including a key/value pair (*add*), removing a pair given its key (*removeKey*) or given its value (*removeValue*), and so on.

It is important to stress that the design of the qualified association with an internal qualifier only works if the qualifier attribute is immutable. If this was not the case, the attribute could change and that change would not necessarily propagate to the value that is the key for the association. This would create an inconsistency. That constraint, however, does not apply to maps with external qualifiers, because in that case the qualifier is not an attribute of the qualified class.

Figure e10.6 presents a map with an external qualifier. The corresponding code is shown below:

```
CLASS Customer
  PRIVATE ASSOCIATION VAR phones:MAP<PhoneType,Phone>
  METHOD getPhones():SET<Phone>
    RETURN phones.getValues()
  END METHOD
  METHOD getPhone(aType:PhoneType):Phone
    RETURN phones.atKey(aType)
  END METHOD
```

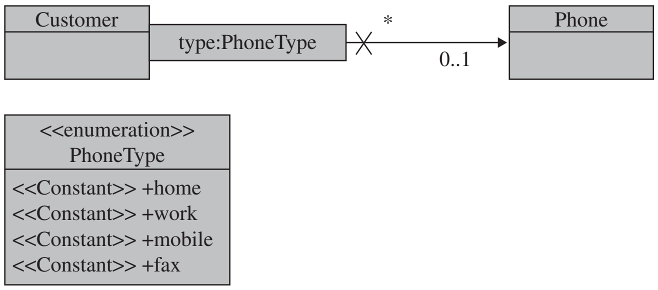

FIGURE e10.6

Qualified association with an external qualifier.

```
    METHOD addPhone(aType:PhoneType;aPhone:Phone)
      phones.add(aType,aPhone)
    END METHOD
    METHOD removePhone(aPhone:Phone)
      phones.removeValue(aPhone)
    END METHOD
    METHOD removePhone(aType:PhoneType)
      phones.removeKey(aType)
    END METHOD
  END CLASS
```

In the case of Figure e10.6 there is an issue to be considered: as the origin of the association has no multiplicity restriction, and the qualifier is not an attribute of the class, nothing assures that the same phone is not associated to two or more keys. For example, the following sequence of commands would produce a phone associated to two different types:

```
aCustomer.addPhone("residential",aPhone)
aCustomer.addPhone("comercial",aPhone)
```

If that is what is expected, nothing else must be said. But if that is not what is meant, the role on the left side should be 1 and the association should be implemented as a bidirectional association, and control mechanisms should be implemented to avoid a phone from being associated to more than one type.

For the next example, we consider that different publishers could edit the same book. Thus we have books and book specifications: a *book* has one single publisher, but a *book specification* has a set of publishers. The ISBN of the *book* for each publisher may be different, but the *book specification* is the same for different publishers, and so is its genre. Figure e10.7 illustrates that a book specification has the genre as an attribute, and therefore each book specification has only one genre that may be associated to different publishers. Figure e10.7 presents a *partition*, that is, a qualified association *to many*, with an internal qualifier. The corresponding code is shown below:

```
CLASS Publisher
  PRIVATE ASSOCIATION VAR bookSpecs:RELATION<BookGenre,BookSpec>
  METHOD getBookSpecs():SET<BookSpec>
```

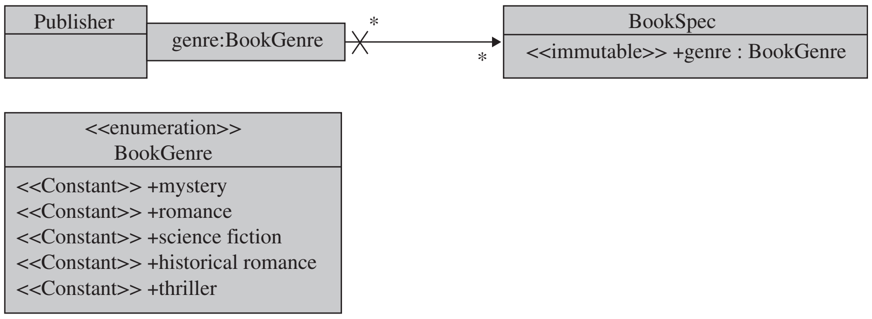

FIGURE e10.7

Qualified association as a partition with an internal qualifier.

```
      RETURN bookSpecs.getValues()
    END METHOD
    METHOD getBookSpec(aGenre:BookGenre):SET<BookSpec>
      RETURN bookSpec.atKey(aGenre)
    END METHOD
    METHOD addBookSpec(aBookSpec:BookSpec)
      bookSpecs.add(aBookSpec.getGenre(),aBookSpec)
    END METHOD
    METHOD removeBookSpec(aBookSpec:BookSpec)
      bookSpecs.removeValue(aBookSpec)
    END METHOD
  END CLASS
```

The implementation above uses a data structure named *RELATION*, which does not exist in most programming languages. But it can be easily implemented. It has an interface similar to *MAP*, but it allows the same key to be associated to many values and not just one. In the example above, for each *BookGenre* the structure associates a set of instances of *BookSpec*.

There is an issue here too. The qualifier attribute *genre* in class *BookSpec* must be *immutable* because the qualifier is internal. If this was not the case, problems like the one aforementioned for the qualified map could jeopardize the design. The attribute is not *unique* because different book specifications may have the same genre.

Does the design presented in Figure e10.7 allow a book specification to be linked to a publisher or different publishers with different genres? The answer is *no*. As the genre is immutable and the *addBookSpec* method takes the key from the *BookSpec* attribute *genre*, it is only possible to add a book specification once to the publisher. As the genre is immutable, the book specification should never be added again with a different genre.

This would not be the case if the genre were not immutable: the same book specification could change its genre and be linked again to one or another publisher. In that case, an inconsistent link to the book specification with the old genre would be left behind.

10.3.4 Unidirectional association with association class

When the association has an association class, it is necessary to implement the creation and destruction of instances of that class each time a corresponding link is added or removed.

Association classes may exist in associations with any multiplicity. However, they are more common and useful in associations that are many to many.

One possible implementation for this kind of association is to create a *map* associating instances of the opposite class to instances of the association class.

Figure e10.8 shows an association class, and its implementation is shown below:

```
CLASS Company
  PRIVATE ASSOCIATION VAR employees:MAP<Person,Job>
  METHOD getEmployee():SET<Person>
    RETURN employees.getKeys()
  END METHOD
  METHOD getJob():SET<Job>
    RETURN employees.getValues()
  END METHOD
  METHOD getJob(aPerson:Person):Job
    RETURN employees.atKey(aPerson)
  END METHOD
  METHOD addEmployee(aPerson:Person)
    employees.add(aPerson,Job.Create())
  END METHOD
  METHOD removeEmployee(aPerson:Person)
    LOCAL VAR aJob:Job
    aJob:=employees.atKey(aPerson)
    employees.removeKey(aPerson)
    aJob.destroy()
  END METHOD
  METHOD removeJob(aJob:Job)
    employees.removeValue(aJob)
    aJob.destroy()
  END METHOD
```

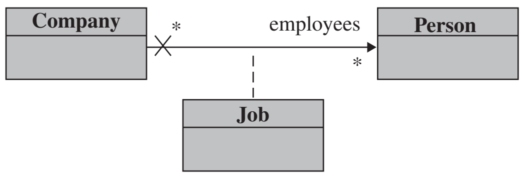

FIGURE e10.8

Unidirectional association with association class.

```
    METHOD replaceEmployee(oldEmployee,newEmployee:Person)
      self.removeEmployee(oldEmployee:Person)
      self.addEmployee(newEmployee:Person)
    END METHOD
  END CLASS
```

In Figure e10.8 we see that when a new link is created from *Company* to *Person*, a new instance of *Job* is automatically created.

The operations that explicitly destroy a job when the link is removed are included in the code to make clear that this is what must be done, as the job cannot exist independently of the link that created it. In languages with a garbage collector and no explicit disposal of objects, it must be ensured that after the link is removed, no other references remain to the object.

The *replace* command only deletes the old link with its job and creates a new one. The instance of *Job* is not maintained when the role is replaced. However, in some cases that could be the meaning intended.

10.4 Bidirectional associations

As mentioned before, the unidirectional implementation of associations is only possible when they are navigable in one direction and there is no multiplicity constraint in the origin role. In other situations, the bidirectional implementation is necessary.

At least three patterns for implementing bidirectional associations have been proposed (Fowler, 2003):

- Implementing the association as two unidirectional associations (*mutual friends* pattern).
- Implementing the association as a unidirectional association in just one of the classes. Navigation would be possible from the opposite direction by means of a query.
- Implementing an intermediary object that represents the association.

In all of the cases above, if the association is navigable in both directions the *get* method must be implemented in both participating classes, because navigation must be allowed in both directions. However, if we have the case of a unidirectional association with a multiplicity restriction at the origin, then the *get* method must be implemented only at the origin class.

The *add* and *remove* methods, if required, may be implemented only in one of the classes, because if they exist in both classes they would be redundant as they would do exactly the same thing.

10.4.1 Mutual friends

The option for implementing bidirectional associations in both directions is the most efficient in terms of time, but it is less efficient in terms of space allocation because each association is implemented twice. It may also require more control overhead, because the implementation of the association in both classes must be synchronized.

The implementation of the unidirectional components of the association follows the recommendations given in Section 10.3. However, three subcases still have to be considered here:

- Both roles are optional.
- Only one role is mandatory.
- Both roles are mandatory.

10.4.1.1 Both roles optional

If both roles are optional and no other constraints are present, then links may be added and removed at will. As links must be added and removed from both sides of the association, auxiliary methods would be needed to add and remove the individual unidirectional links.

These auxiliary methods must not be called in any other place except the *add* and *remove* methods. That is why they must be declared as *private*, but the opposite class must have access to them; thus, they are exported exclusively to that class.

Figure e10.9 presents a bidirectional many-to-many association. The corresponding implementation code is shown below:

```
CLASS Customer
  PRIVATE ASSOCIATION VAR wishes:SET<Book>
  METHOD getWishes():SET<Book>
    RETURN wishes.protected()
  END METHOD
  METHOD addWish(aBook:Book)
    self.privateAddWish(aBook)
    aBook.privateAddWisher(self)
  END METHOD
  METHOD removeWish(aBook:Book)
    self.privateRemoveWish(aBook)
    aBook.privateRemoveWisher(self)
  END METHOD
  PRIVATE METHOD privateAddWish(aBook:Book)
  EXPORTED TO: Book
    wishes.add(aBook)
  END METHOD
```

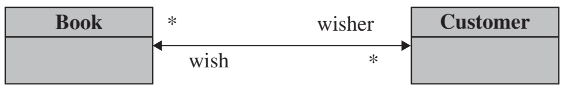

FIGURE e10.9

Bidirectional association from many to many.

```
    PRIVATE METHOD privateRemoveWish(aBook:Book)
    EXPORTED TO: Book
      wishes.remove(aBook)
    END METHOD
  END CLASS

  CLASS Book
    PRIVATE ASSOCIATION VAR wishers:SET<Customer>
    METHOD getWishers():SET<Customer>
      RETURN wishers.protected()
    END METHOD

    PRIVATE METHOD privateAddWisher(aCustomer:Customer)
    EXPORTED TO: Customer
      wishers.add(aCustomer)
    END METHOD

    PRIVATE METHOD privateRemoveCustomer(aCustomer:Customer)
    EXPORTED TO: Customer
      wishers.remove(aCustomer)
    END METHOD
  END CLASS
```

The auxiliary methods in the code above are declared as private but exported to the other participating class.

As mentioned before, the access (*get*) methods must be implemented in both classes if the association is navigable in both directions. But the *add* and *remove* methods may be implemented in just one class. In the example above, they are implemented only in *Customer* class.

10.4.1.2 Only one mandatory role

If one of the roles is mandatory, as for example 1 to *, then the team must ask if the mandatory role is immutable or not. For example, a payment has one order and cannot change to another order (it is immutable), but a car has an owner and can change to another owner if it is sold.

Let us first examine the case of an association link that can be updated, as shown in Figure e10.10.

The code for implementing the classes in Figure e10.10 might be like the following:

```
  CLASS Car
    PRIVATE ASSOCIATION VAR owner:Person
    CONSTRUCTOR METHOD Create(anOwner:Person)
```

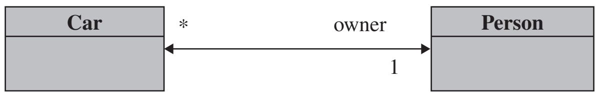

FIGURE e10.10

A bidirectional association with one mandatory role that can be updated.

```
        owner: = anOwner
        anOwner.privateAddCar(self)
    END CONSTRUCTOR METHOD
    METHOD getOwner():Person
        RETURN owner
    END METHOD
    METHOD replaceOwner(anOwner:Person)
        owner.privateRemoveCar(self)
        anOwner.privateAddCar(self)
        owner: = anOwner
    END METHOD
END CLASS
CLASS Person
    PRIVATE ASSOCIATION VAR cars:SET<Car>
    METHOD getCars():SET<Car>
        RETURN cars.protected()
    END METHOD
    PRIVATE METHOD privateAddCar(aCar:Car)
    EXPORTED TO: Car
        cars.add(aCar)
    END METHOD
    PRIVATE METHOD privateRemoveCar(aCar:Car)
    EXPORTED TO: Car
        cars.remove(aCar)
    END METHOD
END CLASS
```

As we can see, the *Car* class implements the only public updating method, which is *replaceOwner*. An ownership link is created every time a new instance of *Car* is created, as defined in the car's constructor. Then, the ownership may only be changed from one person to another. The *Person* class only implements the private methods to add and remove a car from its own local collection.

When a car owner is replaced, it is done in three steps: first the car is removed from the old owner, then it is added to the new owner, and finally the owner is updated to be the new owner.

The second case to be considered here is when the role is immutable. This is the case of a 1 to * association from *Customer* to *Order*, for example: it is mandatory for *Order* and an order can never change its customer. This situation is shown in Figure e10.11, and the corresponding code follows:

```
CLASS Order
    PRIVATE ASSOCIATION VAR customer:Customer
    CONSTRUCTOR METHOD create(aCustomer:Customer)
```

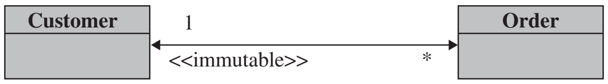

FIGURE e10.11

A bidirectional association with one mandatory immutable role.

```
      customer: = aCustomer
      aCustomer.privateAddOrder(self)
    END CONSTRUCTOR METHOD
    METHOD getCustomer():Customer
      RETURN customer
    END METHOD
  END CLASS
  CLASS Customer
    PRIVATE ASSOCIATION VAR orders:SET<Order>
    METHOD getOrders():SET<Order>
      RETURN orders.protected()
    END METHOD
    METHOD removeOrder(anOrder:Order)
      orders.remove(anOrder)
      anOrder.destroy()
    END METHOD
    PRIVATE METHOD privateAddOrder(anOrder:Order)
    EXPORTED TO: Order
      orders.add(anOrder)
    END METHOD
  END CLASS
```

In this case, an order is immediately associated to a customer when it is created. If an order is removed from a customer then it must be destroyed.

10.4.1.3 Two mandatory roles

There are situations when the two roles of an association are mandatory. In this case, no object can exist without being linked to another. This means that the creation of the links must happen in the constructor of both classes. Figure e10.12 shows an example of a 1 to 1..* association; it is mandatory on both ends.

Once more, the methods that will be implemented depend on deciding if the roles are immutable or not. The customer role from the point of view of an address is immutable. This means that an address cannot change from one customer to another.[3]

[3]In fact, in real life an address could change from one customer to another if we consider that a customer could move to the house of another customer whose address is already registered. However, usually systems do not treat addresses in such a precise way: pragmatically, if a customer moves, it is always a new address that is registered.

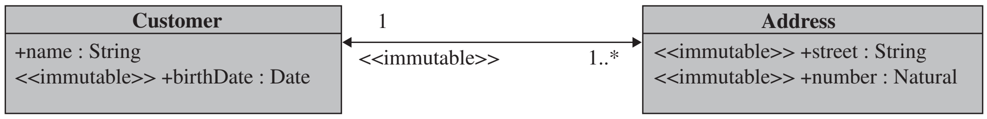
Customer
+name : String
<<immutable>> +birthDate : Date
1
<<immutable>>
1..*
Address
<<immutable>> +street : String
<<immutable>> +number : Natural

FIGURE e10.12

A bidirectional association that is mandatory on both ends.

When a customer is created it must be created with at least one address. This means that the creator of a customer should receive all the data needed to instantiate that address (only *street* and *number* in the example, for simplification).

An existing customer may add a new address, but the method that adds an address cannot simply receive an instance of address as an argument because no address may exist without being linked to a customer. Thus, the customer should implement a method to add a new address that receives the data necessary to instantiate such an address. The code for implementing this situation is shown below:

```
CLASS Address
  PRIVATE ASSOCIATION VAR customer:Customer
  PRIVATE ATTRIBUTE VAR street:String
  PRIVATE ATTRIBUTE VAR number:Natural

  CONSTRUCTOR METHOD Create(aCustomer:Customer;
    aStreet:String; aNumber:Natural)
    customer:=aCustomer
    street:=aStreet
    number:=aNumber
  END CONSTRUCTOR METHOD

  METHOD getCustomer():Customer
    RETURN customer
  END METHOD
END CLASS

CLASS Customer
  PRIVATE ASSOCIATION VAR addresses:SET<Address>
  PRIVATE ATTRIBUTE VAR name:String
  PRIVATE ATTRIBUTE VAR birthDate:Date

  CONSTRUCTOR METHOD create(aName:String; aBirthDate:Date;
    aStreet:String; aNumber:Natural)
    name:=aName
    birthDate:=aBirthDate
    addresses:=Set.new()
    addresses.add(Address.Create(self,aStreeet,aNumber))
  END CONSTRUCTOR METHOD
  METHOD getAddresses():SET<Address>
    RETURN addresses.protected()
  END METHOD
  METHOD addAddress(aStreet:Street; aNumber:Natural)
    addresses.add(Address.create(self,aStreet,aNumber))
  END METHOD
  METHOD removeAddress(anAddress)
   IF address.size()>1 THEN
    addresses.remove(anAddress)
    anAddress.destroy()
   ELSE
    Exception.throw('Customer must have at least one address')
   ENDIF
  END METHOD
  METHOD getName():String
    RETURN name
  END METHOD
  METHOD getBirthDate():Date
    RETURN birthDate
  END METHOD
  METHOD setName(aName:String)
    name:=aName
  END METHOD
END CLASS
```

If the role from *Address* to *Customer* was not immutable, then a *replaceCustomer* command could be implemented as well in the *Address* class. But that is not the case.

10.4.2 Unidirectional implementation

Even if the association is bidirectional, it may be the case that navigation occurs much more often or is more critical in just one direction. If that happens, an option is to implement the association physically in *only one direction*, and implement the *get* method for the opposite side as a search query. The advantage is that the code is simpler, faster in one direction, and space saving. The disadvantage is that the navigation from the opposite direction would be much slower.

This form of implementation is also only possible when the role on the origin of the implemented direction has no multiplicity restriction. If there is a restriction, then the unidirectional implementation is possible only if additional mechanisms to control multiplicity at the origin of the association are implemented, with loss of performance. Below there is an example of unidirectional implementation for the association of Figure e10.10, where the association is physically implemented only in the *Car* class:

```
CLASS Car
  PRIVATE ASSOCIATION VAR owner:Person
  CONSTRUCTOR METHOD Create(anOwner:Person)
    owner: = anOwner
  END CONSTRUCTOR METHOD
  METHOD getOwner():Person
    RETURN owner
  END METHOD
  METHOD replaceOwner(newOwner:Person)
    owner: = newOwner
  END METHOD
END CLASS
CLASS Person
  METHOD getCars():SET<Car>
    LOCAL VAR cars:SET<Car>
    cars: = Set.new()
    FOR EACH car IN Car.getAllInstances() DO
      IF car.getOwner() = self THEN
        cars.add(car)
      END IF
    END FOR
    RETURN cars
  END METHOD
END CLASS
```

The *Car* class above is implemented just like if the association was unidirectional from it to *Person*. The *Person* class implements only the *getCars* method through a search query on the set of all instances of *Car*. This implementation only works well if the role multiplicity at the origin has is unconstrained.

Regarding the time complexity of the *get* methods, the bidirectional implementation runs in *constant time*[4] in both directions, and the unidirectional implementation has constant time for *getOwner* and *linear time*[5] for *getCars*, that is, the performance of the *getCars* method depends on the number of cars registered. This design may be optimized by using *hash* techniques to index the set of customers relative to their orders. This way, the complexity of the *getCars* query may almost become constant in practice with the cost of some extra memory space.

A limitation of this technique is that the programming language must provide a method to access all instances of a class. Otherwise, the programmer must supply such a mechanism. Unfortunately, this is a hazardous design that may cause trouble due to the global visibility of those instances (Cardoso, 2011).

[4]Time remains the same even if the number of cars increase.
[5]Time increases as the number of customers increase. Linear time means that time is a function $t(x) = ax + b$, where x is the size of the set for customers and a and b are constants.

10.4.3 Implementation with an intermediary object

A bidirectional association may also be implemented by means of an intermediary object that represents the association. The intermediary object consists of a table with pairs of linked instances. Possible implementation for the bidirectional association from many to one of Figure e10.11 with an intermediary object follows:

```
GLOBAL VAR orderXcustomer:MAP<Order,Customer>
VISIBILITY RESTRICTED TO: Order, Customer

CLASS Customer
   METHOD getOrders():Set<Order>
      RETURN orderXcustomer.getKeysFor(self)
   END METHOD
   METHOD addOrder(anOrder:Order)
      orderXcustomer.add(anOrder,self)
   END METHOD
   METHOD removeOrder(anOrder:Order)
      orderXcustomer.remove(anOrder,self)
   END METHOD
END CLASS
CLASS Order
   CONSTRUCTOR METHOD Create(aCustomer:Customer)
     orderXCustomer.add(self,aCustomer)
   END CONSTRUCTOR METHOD
   METHOD getCustomer():Customer
      RETURN orderXcustomer.atKey(self)
   END METHOD
END CLASS
```

If the programming language allows it, the association should be declared a global variable that is visible only by the participating classes. Unfortunately, most commercial languages would not allow such feature and the association would be implemented as a global variable with no visibility restriction.

If the association was many to many instead of one to many, then the *MAP* data type used above should be replaced by *RELATION* with small adjustments in the getters and setters.

The intermediary object approach tends to be much simpler and maintainable than the former ones. It also mimics the structure of a relational database because associations, as seen in Chapter 13, are implemented as intermediary tables, which correspond to the intermediary object here.

The disadvantage of this method is that getters are slower in both directions when compared to the bidirectional approach and the global visibility of the intermediary object may cause design hazards if not used carefully.

10.5 Delegated methods and system operations

To this point, we have shown how to generate code for classes, attributes, associations, and their corresponding basic methods that may be considered part of their basic structure. Now it is the time to explain how to implement delegate methods and system operations. These should be implemented by following the dynamic models explained in Chapter 9.

System operations are implemented as a sequence of messages labeled 1, 2, 3, . . ., *n* that are sent by the controller. A delegate method labeled with *x* in the interaction diagram should be implemented by a sequence of messages labeled *x*.1, *x*.2, *x*.3, *x*.*n* that are sent by the object that receives the message labeled with *x*.

Let us look again at Figure 9.43, reproduced here as Figure e10.13.

The *add2Cart* system command should therefore be implemented as a sequence of messages labeled with 1, 2, and 3. The corresponding pseudocode could be something like the following:

```
CLASS Livir
  ASSOCIATION VAR carts:MAP<CartId,Cart>
  ASSOCIATION VAR books:MAP<Isbn,Book>
  METHOD add2Cart(aCartId:CartId; anIsbn:Isbn;
    aQuantity:Natural)
    LOCAL VAR aCart:Cart
    LOCAL VAR aBook:Book
    aCart:=carts.at(aCartId)           --message 1
    aBook:=books.at(anIsbn)            --message 2
    aCart.insertItem(aBook,aQuantity)  --message 3
  END METHOD
  ...
END CLASS
```

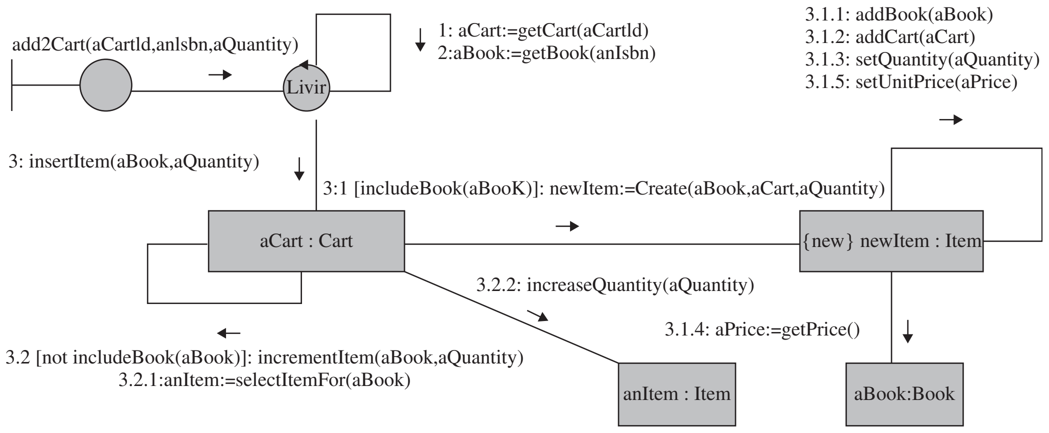
add2Cart(aCartld,anlsbn,aQuantity)
Livir
1: aCart:=getCart(aCartld)
2:aBook:=getBook(anIsbn)
3.1.1: addBook(aBook)
3.1.2: addCart(aCart)
3.1.3: setQuantity(aQuantity)
3.1.5: setUnitPrice(aPrice)
3: insertItem(aBook,aQuantity)
3:1 [includeBook(aBooK)]: newItem:=Create(aBook,aCart,aQuantity)
aCart : Cart
{new} newItem : Item
3.2.2: increaseQuantity(aQuantity)
3.1.4: aPrice:=getPrice()
3.2 [not includeBook(aBook)]: incrementItem(aBook,aQuantity)
3.2.1:anItem:=selectItemFor(aBook)
anItem : Item
aBook:Book

FIGURE e10.13

Reference communication diagram.

The implementation of the *insertItem* delegate method in the *Cart* class consists of the sequence of messages 3.1 and 3.2, because *insertItem* is labeled with 3. These messages are conditional and mutually exclusive. Thus, their implementation corresponds to an *if-then-else-endif* structure, as shown below:

```
CLASS Cart
  ...
  METHOD insertItem(aBook:Book,aQuantity:Natural)
    LOCAL VAR newItem:Item
    IF self.includeBook(aBook) THEN
      newItem:=Item.Create(aBook,self,aQuantity)  --3.1
    ELSE
      self.incrementItem(aBook,aQuantity)         --3.2
    ENDIF
  END METHOD
END CLASS
```

Notice that the variable name *aCart* is not known in this method and must be replaced here by *self*, which stands for the instance of *Cart* that is performing the method.

The *incrementItem* delegate method in the *Cart* class is labeled with 3.2. Therefore, its implementation is the sequence of messages 3.2.1 and 3.2.2:

```
CLASS Cart
  ...
  METHOD incrementItem(aBook:Book; aQuantity:Natural)
    LOCAL VAR anItem:Item
    anItem:=self.selectItemFor(aBook)  --3.2.1
    anItem.increaseQuantity(aQuantity)  --3.2.2
  END METHOD
END CLASS
```

The constructor of the *Item* class is complex and as it is labeled with 3.1, its implementation consists of the sequence of messages 3.1.1 to 3.1.5. The *Item* class would look like this:

```
CLASS Item
  CONSTRUCTOR METHOD Create(aBook:Book; aCart:Cart;aQuantity:Natural)
    LOCAL VAR aPrice:Money
    self.addBook(aBook)        --3.1.1
    self.addCart(aCart)        --3.1.2
    self.setQuantity(aQuantity) --3.1.3
    aPrice:=aBook.getPrice()   --3.1.4
    self.setPrice(aPrice)      --3.1.5
  END CONSTRUCTOR METHOD
END CLASS
...
```

The other methods that appear in Figure e10.13 are basic ones and must be implemented following their default definition.

10.6 Patterns for filtered queries

In the examples shown to this point, basically two kinds of *get* queries were implemented: those that return all objects linked by a role, and those that return a single object given its identification, such as a qualifier or position.

However, sometimes it will be necessary to obtain a subset of objects in a given role. That subset is usually obtained by a filter such as "customers older than 30 years old," "orders between 100 and 200 dollars," "orders above 100 dollars issued last week," etc.

There are at least three patterns to deal with this diversity of queries (Fowler, 2003):

- Implement a *single query* that returns all objects, and let the object that needs the information apply a filter to that collection.
- Implement *specific queries*, each one with a different filter, so that the object that needs the information calls a specific method in each case.
- Implement a generic query that uses a *filter object*.

The single query and filter object approaches imply the implementation of a single method that leaves the class that has the responsibility simpler. However, it requires more code to be written in the classes that need the information.

The specific queries approach has as a consequence a greater number of methods in the class that holds the responsibility. But the classes that need the information would make simple calls and receive the information without any need for postfiltering.

The choice of one pattern or another must be made by the designer depending on the number of possible filters and potential calls. If there are only a few possibilities for filtering and many calls, the best choice is specific queries. If the quantity of filters is high and there are only a few calls for each filter, then the single query or filter object approach would be better. Single query is more straightforward, but produces more work during the coding phase: each time a filter is used, the respective code must be implemented in the class that calls the query. Filter object requires defining a class that implements the *filter object*, but after that investment is done, filter object tends to be simpler than single query.

A *filter object* is a parameter that is passed to the general query method. The filter object must contain attributes and associations to other objects. Those attributes and associations are used by the query to decide which objects must be returned.

For example, let us revisit the definition of the *Book* class shown in Figure e10.14.

Suppose now that we need to query books filtered by *title*, *authorsName*, or *price*.

Applying the single query approach, a *getBooks*() method should be implemented in the *Livir* class, which is responsible for returning the set of all books. If another object needs to filter that set, it must apply the filter to the resulting set.

If the specific queries approach is used, the *Livir* class should implement three queries that may be considered variants of the base query:

- *getBooksByTitle*(*aTitle:String*).
- *getBooksByAuthorsName*(*anAuthorName:String*).
- *getBooksByPrice*(*aRange:Range<Money>*).

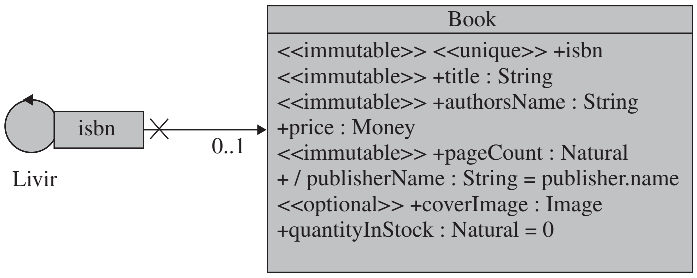
Book
<<immutable>> <<unique>> +isbn
<<immutable>> +title : String
<<immutable>> +authorsName : String
+price : Money
<<immutable>> +pageCount : Natural
+ / publisherName : String = publisher.name
<<optional>> +coverImage : Image
+quantityInStock : Natural = 0
isbn
0..1
Livir

FIGURE e10.14

Reference class for queries.

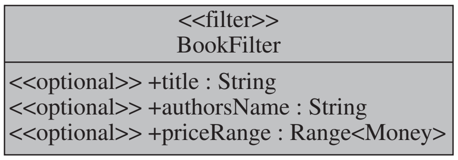
<<filter>>
BookFilter
<<optional>> +title : String
<<optional>> +authorsName : String
<<optional>> +priceRange : Range<Money>

FIGURE e10.15

A class for a filter object.

Each of these queries return only the elements that satisfy the respective filters indicated as arguments. If a combination of criteria is needed, then new queries usually should be created for the combined criteria.

The filter object approach requires the definition of a class to represent the filter object. For the example, a *BookFilter* class could be defined as shown in Figure e10.15.

In this case, the *Livir* class should implement a single *getBooks*(*aBookFilter*:*BookFilter*): *Set* < *Book* > method for querying the set of all books.

Using this method, a query to return the books written by "Douglas N. Adams" that cost less than 10 dollars could be written like this:

```
...
myFilter: = BookFilter.Create()
myFilter.setAuthorsName("Douglas N. Adams")
myFilter.setPriceRange(Range(US$0,US$10))
cheapAdamsBooks: = self.getBooks(myFilter)
...
```

As a result, the variable *cheapAdamsBooks* contains the set of all instances of *Book* whose author is "Douglas N. Adams" and that cost less than 10 dollars. As the attribute *title* of *BookFilter* was not filled for the query, it is ignored, and any title would qualify for the filter.

10.7 The process so far

	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.	
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 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.

(*Continued*)

(Continued)

	Inception	Elaboration
		• 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. 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.
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		
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.	

10.8 Questions

1. Explain in detail how a delegate message that appears in a communication diagram must be implemented.
2. What are the basic operations that must be implemented for most kinds of associations? Which ones must be implemented for specific kinds of associations?
3. Propose a generic implementation for the query *getBooks*(*aBookFilter*:*BookFilter*): *Set*<*Book*>. Try to keep it as general and reusable as possible.
4. Propose an implementation for the system commands and delegate methods presented in Figures 9.42 and 9.44.