Chapter 5

Domain Modeling Language

At least since the introduction of the Chen Entity-Relationship Model [Chen, 1976], well before object-oriented programming became mainstream, relations are an integral part of domain modeling languages. Therefore, we would expect that recent programming languages should have constructs to facilitate the implementation of relations. In fact, about twenty years ago, Rumbaugh argued convincingly for the inclusion of relations as a primitive declarative construct in object-oriented programming languages:

> Object-oriented languages express *classification* (the grouping of objects into classes) and *generalization* (the refinement of classes into subclasses) well, but do not contain syntax or semantics to express relations directly. Any program can implement particular relations on an ad hoc basis, but the abstraction may get lost in the implementation mechanisms.
> [...]
> Object-oriented languages have built-in constructs for generalization because it is a natural concept that people use in ordinary discourse; it allows algorithms to be written more concisely and more clearly; and it is common enough to justify building it into a language. Relations are also natural, productive, and common in abstracting applications. An object-oriented language is more expressive if relations are a primitive declarative construct, on the same footing as classes.
>
> [Rumbaugh, 1987]

And yet, surprisingly, none of the more recently developed object-oriented programming languages provide such a construct. Unfortunately, the consequence of this oversight in the design of new programming languages is an undesirable burden for the programmers that need to implement domain models, as we have seen in Section 3.3.

Thus, given the current state of the object-oriented programming languages, a possible approach to simplify the implementation of domain models is to extend those languages with a set of new constructs for implementing relations. This solution, however, has the inconvenient that interferes with the development tools that programmers use and goes,

therefore, against the guiding-principles that I established in Section 1.2.2.

The approach that I propose in this chapter follows a different route. Rather than extending a programming language, I propose a new language—the Domain Modeling Language (DML)—that complements and integrates with the Java programming language. The DML language is a micro-language designed specifically to implement the structure of a domain model: It has constructs for specifying both entity types and associations between entity types. Associations are the primitive declarative construct for relations that was argued for by Rumbaugh [1987].

I start with the rationale underlying the development of this new language, and then I describe the language in detail, showing both its syntax and how it integrates with the Java language. I discuss related work at the end of the chapter, in Section 5.8.

5.1 DML's Rationale

The objects of a domain model have special needs, compared to the remaining objects of an application, as we have discussed already in Section 2.1.4 and Section 2.1.5. These special needs may include, for example, that they are stored in a database whenever they change, or that they be accessed by multiple concurrent threads in a consistent way. Most often, because of these needs, the classes that represent the domain objects must follow some coding conventions, to ensure that their instances behave as expected. For instance, if we are using the JVSTM described in Section 4.4 to make the domain objects transactional, we must use instances of the class `VBox` for all the mutable fields of a domain class.

In some cases, as when using the JVSTM, the code conventions are simple to follow. But, as simple as they may be, they still must be applied manually by the programmers, introducing unnecessary effort into the programming task, and opening the possibility of errors. Thus, the idea of automating these programming tasks was the first incentive that led me to the development of the DML: I wanted a simple way to specify that a class is a domain class, and to have this class transformed automatically into a class that uses versioned boxes for all its fields.

Nevertheless, this reason alone is not sufficient to justify the need of a new language. In fact, transforming each of a class's field into a field of another type and changing the code that accesses that field to use a different access expression, is well within the reaches of Java's annotations and post-processing technologies.

The most important reason for the development of the DML language, however, was the lack of support in Java for the implementation of associations between classes. To solve this problem in a convenient way, I would need to extend the syntax of the Java language

with new top-level syntactic constructs to represent associations. Unfortunately, the extension mechanisms available in Java do not allow syntax extensions. The top-level constructs of a Java program are the class and the interface, and no provisions exist to create other syntactic constructs. So, I decided to create a new language—the DML—with the appropriate constructs to represent both the classes and the associations of a domain model.

Yet, it is not the goal of DML to replace Java. Rather, it should integrate with Java, so that programmers can still leverage on all the advantages of the Java programming language. The key idea is that this new language should be as small as possible, by providing constructs to represent a domain model's structure, but leave all the rest to Java. In particular, the behavior of the domain model's classes should be programmed in Java, as before. Furthermore, the code that implements the classes' behavior should be the same, regardless of how the class structure is developed (either with DML or in plain Java).

Therefore, DML is designed to represent only the structural aspects of a domain model, using, for that, constructs that are as close as possible to the constructs used at the modeling level—constructs such as *class* and *association*. But also, because the domain model is not complete without its behavior, which is programmed in Java, the model's structure expressed in DML must integrate with the Java code that implements the model's behavior. Both the syntax and the semantics of DML reflect this requirement of seamless integration.

As the DML is designed to target specifically Java programmers, its syntax borrows from the syntax of Java whenever possible. Moreover, the semantics of DML is specified by describing the set of Java classes that result from a DML domain specification: To implement the domain model's behavior, programmers must know the interfaces of these resulting classes. Thus, implicit to the semantics of DML is that there is a compilation process that transforms a DML domain specification into a set of Java classes.

Finally, the last requirement that influences the design of DML is that, even though programmers must know the interfaces of the classes generated by the DML compiler, they should not need to know the implementation details of those classes. In fact, to avoid round-trip problems, the source code of the class generated by the DML compiler should not be accessible to the programmers: If programmers could modify the code generated by the DML compiler, the DML compiler would not be able to regenerate the code when some part of the DML domain specification changes.

In Figure 5.1, I illustrate the expected effect of using the DML in the tool chain typically used for the development of Java applications. The DML compiler is the only element that is new. It reads a set of DML source files, which specify the structure of a domain model, and generates a set of Java source files that implement the structure of that

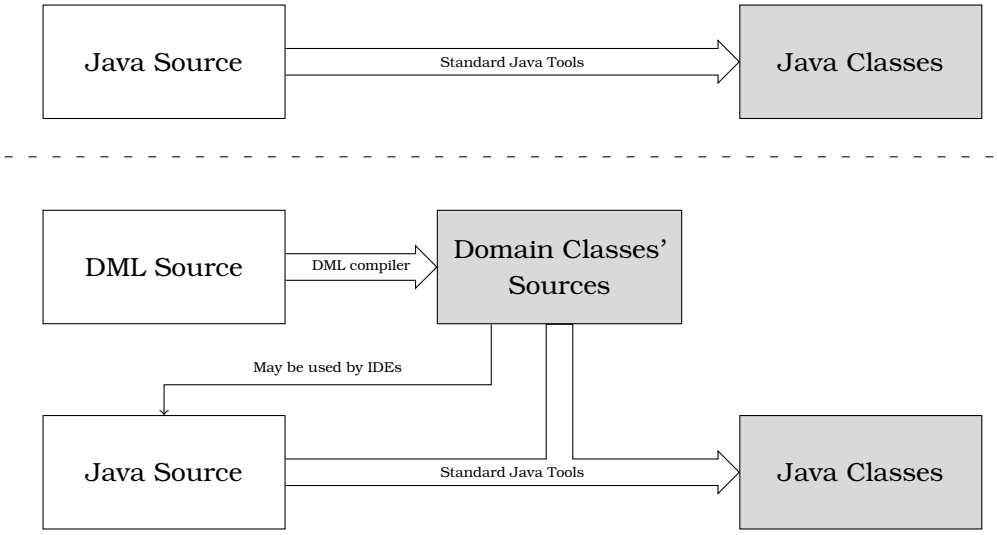

Figure 5.1: The effect that the DML has on the tool chain typically used for Java application development. The dashed line separates the two different scenarios of tool chain usage. The scenario above the line corresponds to the situation where the DML is not used. The scenario below the line illustrates the changes caused in the tool chain by using the DML. Rectangles represent implementation artifacts of the development process. The rectangles with a white background represent the artifacts that are edited by the programmers, whereas the rectangles with a gray background are tool-generated artifacts that the programmers never modify manually. The large arrows represent the transformation of one kind of artifacts into another kind of artifacts, performed by the execution of the development tools indicated inside the arrow. Finally, the single arrow pointing from the *Domain Classes' Sources* to the *Java Source* indicates that the former artifacts may be used by IDEs to help programmers in the development of the latter.

domain model. I decided to generate Java source code, rather than compiled classes, to eliminate dependencies between this transformation process and other Java classes. As we shall see, the Java source code generated by the DML compiler may need additional application-specific classes to compile. Yet, the DML compiler itself does not need any other application-specific Java class to perform the transformation. So, by generating only source code, the DML compiler may be executed at any time during the development process.

5.2 Grammar Notation and Lexical Structure

The syntax of the DML language is defined using a context-free grammar that I introduce in the following sections. To present this grammar, I use the notation that is used in the Java Language Specification [Gosling et al., 2005, Chapter 2]:

- Nonterminal symbols are shown in *italic* type, with no spaces between words and with the first letter of each word capitalized. For example, the following are nonter-

minal symbols: *Identifier*, *DomainSpecification*, and *EntityTypeDeclaration*.

- Terminal symbols are shown in `fixed width` font. Examples of terminal symbols are: `class`, `}`, `;`, and `playsRole`.
- Production rules are typeset with the left-hand side nonterminal—the nonterminal symbol defined by the rule—followed by a colon, on a line by itself. Then, on each of the following lines, there is a sequence of symbols that defines an alternative right-hand side of the rule. Thus, the rule

  QualifiedName:
  Identifier
  QualifiedName **.** *Identifier*

  defines the nonterminal *QualifiedName* as either an *Identifier*, or a *QualifiedName*, followed by the terminal **.** and an *Identifier*. This rule is part of the syntactic grammar for the DML language and defines the nonterminal *QualifiedName* as a sequence of one or more identifiers (explained below) separated by dots.
- The suffix $_{opt}$, which may appear on the right-hand side of a rule, indicates that the symbol that precedes the suffix is optional—that is, that in fact the right-hand side corresponds to two alternative right-hand sides, one where the symbol occurs and another where the symbol does not occur.

A lexical grammar for the Java programming language is given in Chapter 3 of the Java Language Specification [Gosling et al., 2005]. This grammar defines how sequences of Unicode characters are translated into a sequence of *input elements*, which may be *white space*, *comments*, or *tokens*. The *tokens*, which consist of *identifiers*, *keywords*, *literals*, *separators*, and *operators*, form the terminal symbols for the syntactic grammar of Java.

The DML is a much simpler language. For instance, in DML there are no *literals* nor *operators*. DML uses only *identifiers*, a few *keywords*, and *operators*. Yet, because DML must integrate seamlessly with Java, I use the lexical grammar defined for Java to define the lexical structure for the DML language. Thus, the syntax of DML for such things as comments, identifiers, and white space is the same as in Java.

Note that, even though the DML language does not use most of the keywords of Java nor any literals, the identifiers used in a DML specification must follow the restriction that they cannot be a *keyword*, a *boolean literal*, or the *null literal*, because these identifiers will appear as identifiers in the Java source code generated by the DML compiler.

In the grammar of DML presented in the following sections I use the nonterminal symbol *Identifier* as it is defined in the Java lexical grammar: Informally, as a sequence

DomainSpecification:
    DomainDeclarations^{opt}

DomainDeclarations:
    DomainDeclaration
    DomainDeclarations DomainDeclaration

DomainDeclaration:
    ValueTypeDeclaration
    EntityTypeDeclaration
    AssociationDeclaration

Listing 5.1: Syntactic rules for a domain specification. *DomainSpecification* is the goal symbol for the DML syntactic grammar.

of Unicode characters that start with a letter, and is followed by any number of letters or digits. Furthermore, I use the nonterminal symbol *DecimalNumeral* as it is defined in the Java lexical grammar, also: Either as the digit `0`, or as a sequence of digits starting with a digit from `1` to `9`. Finally, I use the nonterminal *QualifiedName* as defined by the production given above. All the remaining nonterminal symbols are defined in one of the grammar fragments shown in the following sections.

5.3 Domain Specification

The DML language allows us to represent only the structural aspects of a domain model. It has no constructs to describe the behavior of a program. So, what the DML compiler reads and processes should not be called a program. Rather, I call it a *domain specification*.

The syntax of a domain specification is defined by the grammar rules shown in Listing 5.1. The nonterminal symbol *DomainSpecification* is the goal symbol of the DML syntactic grammar.

According to these rules, a **domain specification** is a sequence of zero or more domain declarations, which, in turn, are either a *value type declaration*, an *entity type declaration*, or an *association declaration*. The following sections describe, in detail, the syntax and the semantics of each of these domain declarations.

Each domain declaration introduces a new name that may be used in other parts of the domain specification to refer to the domain element introduced by this declaration. Thus, the only restriction to the order of domain declarations is that the names are introduced before they are used.

ValueTypeDeclaration:
`valueType` *ValueTypeName AliasClause*^{opt} `;`

AliasClause:
`as` *ValueTypeName*

ValueTypeName:
QualifiedName

Listing 5.2: Grammar rules for the syntax of a value type declaration.

5.4 Value Types

In the DML language, I distinguish between value objects and entities, as described in Section 2.2.3. Value objects, unlike entities, are immutable, are not persistent by themselves (only as part of entities), and do not participate in bidirectional associations. Therefore, the code used to implement a value type is significantly different from the code used to implement an entity type.

Furthermore, often the value types used in a domain model are not specific of that domain model. Instead, they may be used across many different domains. So, in many cases value types are provided already by a third-party framework or toolkit. If, however, a value type does not exist yet, the implementation of its structure is mostly trivial, because value objects are immutable.

For these reasons, the DML language does not allow the specification of new value types. Yet, value types are needed to define new entity types: All the attributes of an entity type must be of some value type. That is why the DML language has value type declarations.

New value types are introduced in a domain specification by value type declarations, which follow the syntax specified by the grammar productions shown in Listing 5.2.

Each value type declaration introduces into the domain specification the name of a value type. The definition of this value type, however, is made outside the DML's domain specification.

In its simplest form, a value type declaration is only the keyword `valueType` followed by the fully-qualified name of a new value type. That name becomes a new valid name that may be used wherever a value type is expected. When the *alias clause* is used, the name that is introduced into the domain specification as a new value type is the name that follows the keyword `as`; the real name of the value type is used only by the DML compiler to generate the code with the appropriate types.

```
valueType boolean;
valueType byte;
valueType char;
valueType short;
valueType int;
valueType float;
valueType long;
valueType double;
valueType java.lang.Boolean   as Boolean;
valueType java.lang.Byte      as Byte;
valueType java.lang.Character as Character;
valueType java.lang.Short     as Short;
valueType java.lang.Integer   as Integer;
valueType java.lang.Float     as Float;
valueType java.lang.Long      as Long;
valueType java.lang.Double    as Double;
valueType java.lang.Number    as Number;
valueType java.lang.String    as String;
```

Listing 5.3: Default value types in the DML language. The first eight value type declarations introduce the names of the eight primitive types in Java as value types. The following eight declarations introduce the wrapper reference types, which, because of the alias clause, should be used without the `java.lang` package qualifier. Likewise for the last declaration, which introduces the name `String`.

As an example, I show in Listing 5.3 the value type declarations for the value types defined by default in the DML language.

5.5 Entity Types

Entity types are the basic elements of a domain specification. Each entity type describes the structure of a set of similar entities, which are the objects that represent the domain's state. Entities hold their state in a set of attributes, which may change during the execution of an application as the state of an entity changes. The value of each attribute, however, must be a value object—that is, the type of each attribute must be a value type. Entities may refer to other entities only through the traversal of the associations between their types.

5.5.1 Syntax of Entity Type Declarations

A new entity type is introduced in a domain specification by an *entity type declaration*, which follows the syntax specified by the grammar rules shown in Listing 5.4 on the next page.

EntityTypeDeclaration:
`class` *EntityTypeName Super*^{opt} *EntityTypeBody*

EntityTypeName:
QualifiedName

Super:
`extends` *EntityTypeName*

EntityTypeBody:
`;`
`{` *AttributeDeclarations*^{opt} `}`

AttributeDeclarations:
AttributeDeclaration
AttributeDeclarations AttributeDeclaration

AttributeDeclaration:
ValueTypeName Identifier `;`

Listing 5.4: Grammar rules for the syntax of an entity type declaration.

An entity type declaration is a stripped-down version of a class declaration in the Java programming language. In fact, many entity type declarations in DML are valid class declarations in Java, even though only a few class declarations in Java are valid entity type declarations in DML.

As in Java, after the keyword `class` comes the name of the new entity type. But, whereas in Java the name of the new class must be an *Identifier*, in DML the *EntityTypeName* may be a qualified name.

When the new entity type is a subtype of another entity type, we represent that inheritance relationship by using the keyword `extends` followed by the name of the entity type of which the new type is a subtype. Note that the name after the keyword `extends` must be the name of an entity type introduced by a previous entity type declaration. In DML, an entity type cannot be a subtype of a value type.

The body of an entity type declaration is used in DML only to specify the entity type's attributes, using a syntax similar to the syntax of field declarations in Java. If an entity type does not have new attributes (other than those inherited), the body of its declaration may be replaced by a semicolon. An example of such a simple entity type declaration is

```
class Bank;
```

Yet, in general, entity types have one or more attributes. So, typical examples of entity type declarations are shown in Listing 5.5 on the following page.

```
class Client {
  String name;
}

valueType a.business.api.Money as Money;

class Account {
  Money balance;
}

class ClientAccount extends Account {
  boolean closed;
}
```

Listing 5.5: Examples of entity type declarations in DML. Both `Client` and `Account` are top-level entity types that do not inherit from any other type. The `ClientAccount` type, however, is a subtype of `Account`. Also, the `Account` entity type uses a value type introduced in the previous declaration.

5.5.2 Semantics of Entity Type Declarations

I specify the semantics of a domain specification by prescribing the minimal set of Java classes that a compiler of the DML language must produce when it compiles the domain specification. Also, I prescribe which fields and methods each generated class must have. Different compilers for the DML language may vary in the classes they produce. Yet, all the compilers must conform to the minimal interface specified here, because programmers depend on this interface to develop the rest of the domain model.

Therefore, in this section, I specify the semantics of an entity type declaration by describing the minimal Java interface that a DML compiler must produce when it compiles an entity type declaration.

We have seen in Section 3.3.1 that classes from a UML class diagram are implemented naturally as classes in Java: Typically, each class of a class diagram—corresponding to an entity type in DML—is implemented by a single class in Java.

In DML, however, an entity type is implemented by two classes, as depicted in Figure 5.2 on the next page. Each class implements one of the two aspects of a domain entity:

- The first class—the **state class**—is abstract and implements the domain entity's structural aspects. The DML compiler generates this class from the domain specification. Thus, programmers should not edit this class manually.
- The second class—the **behavior class**—extends the state class and implements the domain entity's behavioral aspects. This class, unlike the state class, cannot be

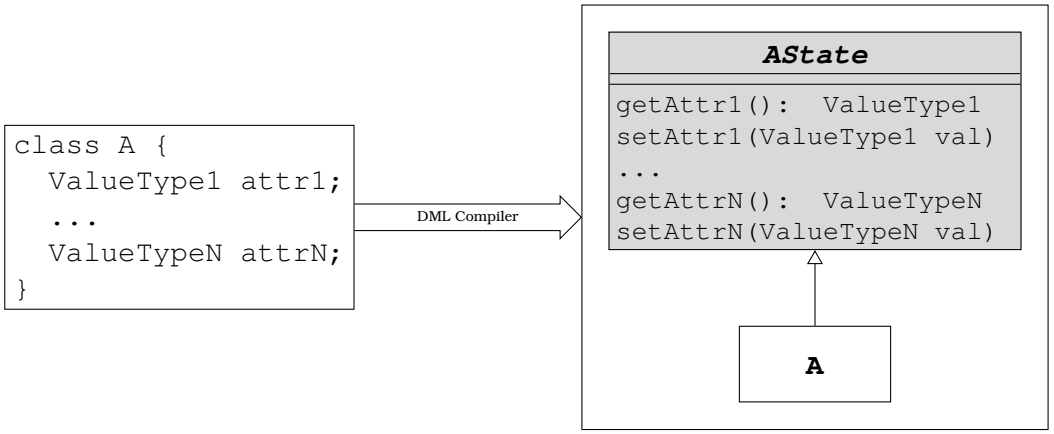

Figure 5.2: The result of compiling an entity type in DML. An entity type `A` is transformed into two Java classes `AState` and `A`. The abstract class `AState` has a getter and a setter for each of the entity type's attributes. The gray background in the class `AState` indicates that the DML compiler generates this class and, thus, that programmers should not edit the class.

generated by the DML compiler, because the domain specification does not have any behavior specification.[1] Instead, the implementation of this class is the responsibility of the domain model programmers: This is the class where programmers implement the behavior of the entity.

The DML compiler uses this compilation strategy of separating a class in two to avoid round-trip problems: Whenever we change the domain specification, we must reexecute the DML compiler to update the classes generated in previous compilations, but, because programmers do not edit the state classes, the DML compiler may regenerate those classes from scratch each time it runs.

The state class that the DML compiler generates from an entity type has both a getter method and a setter method for each of the entity type's attributes. The DML compiler forms the names of these methods by concatenating the prefixes `get` and `set`, respectively, with the name of the attribute, after capitalizing the first letter of the attribute. So, given an entity object `obj`, the method call `obj.getAttrName()` returns the `obj`'s value for the attribute named `attrName`, whereas the call `obj.setAttrName(newVal)` sets the value of the attribute to the value `newVal`. No other methods can access the attribute.

Even though a state class has all the attributes of an entity type and no abstract method, it must be abstract, because it does not represent an entity. An entity contains both state and behavior, but the instances of a state class contain only the state. Therefore, the state class is abstract to prevent the creation of instances of an incomplete type.

[1] In fact, the DML compiler generates an empty behavior class if the class does not exist yet, so that we may compile the domain model without errors. If the class exists, however, the DML compiler does not modify it.

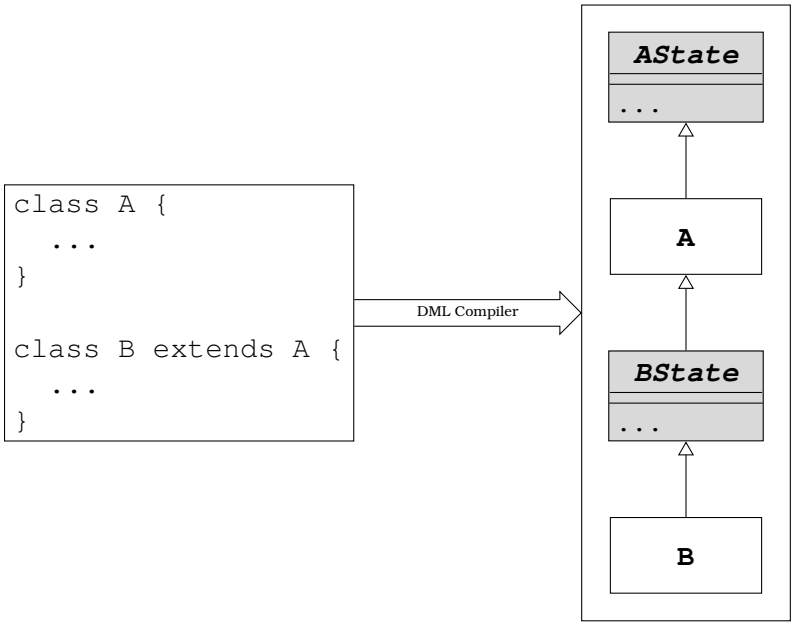

Figure 5.3: The result of compiling an hierarchy of entity types in DML. The Java state class that results from the compilation of an entity type extends the behavior class that corresponds to its supertype: In this case, the class `BState` extends `A`.

The class that represents both the state and the behavior of an entity is the behavior class, which implements the behavior and inherits the state from the state class. Note that the methods in the behavior class may access the state of an instance by using the getter and the setter methods inherited from the state class.

As a matter of fact, the state class is meant to be used only as the superclass of its corresponding behavior class. No other classes should extend it. Also, no fields or methods or any other part of the code should refer to a state class as a type. Instead, all the remaining code should use the types that correspond to behavior classes. For instance, in Figure 5.3, I show that, if `B` is a subtype of `A`, the class that implements the structure of `B`, the class `BState`, must extend the class `A`, rather than the class `AState`. If the class `BState` extended the class `AState` instead of `A`, it would not inherit the behavior of `A`, as expected.

Finally, there is only one restriction regarding the implementation of a behavior class—that it should not have any state of the entity type; the state belongs in the state class. Besides that restriction, programmers are free to implement the behavior class in whatever way they want. For instance, programmers may make the class abstract, if it is not meant to have any instances; they may make the class implement any number of interfaces; or they may add the methods they need to implement the entity's behavior. In particular, they may override any of the methods that are specified in this section, which the behavior class inherits from the state class.

5.6 Associations

In UML, relations are called *associations* and they may connect two or more classes to represent a relationship that exists among the objects that the connected classes describe. The DML language uses the name *association*, also, to represent relationships. But, unlike UML, DML does not allow the specification of associations with an arbitrary arity; in DML all associations must be binary. I chose to support only binary associations in the DML language mostly for pragmatical reasons.

First, because even though the semantics of a generic n-ary relation is naturally defined in mathematical terms as a set of tuples, the semantics of n-ary relationships as a domain modeling construct is either ill-defined, or confusing and error-prone [Génova, Llorens, and Martínez, 2002].

Second, because the implementation of n-ary associations in the object-oriented programming paradigm is not that simple. In an object-oriented program, programmers use binary associations to navigate in the object graph, going from one object to another by traversing a link—that is, an instance of an association—between the two objects. Typically, programmers implement such binary links in an object-oriented domain model as references or pointers from one object to the other. When the association is not binary, however, to find the object at the other end of a link, we need more than an object: We need all the objects in the link but one. Thus, we no longer can use simple references between objects to implement such associations. But this is more of a concern for the implementation of the DML compiler, which would need to generate the code to implement the correct semantics (provided that we could agree on one). If we ignore the implementation problems raised by non-binary associations, however, there are still usage problems: Traversing a binary link in a typical implementation of an object-oriented domain model may be linguistically quite different from traversing links that relate more than two objects.

The third pragmatical reason for limiting the DML to binary associations, was the observation that associations among three or more entity types are seldom used in the design or in the implementation of a domain model: Most probably because of the two reasons above, programmers shy away from n-ary associations when they develop their domain models.

The fourth and final reason is that limiting the DML to binary associations does not prevent the implementation of a domain model. In the rare cases where programmers use a non-binary association, it is possible to replace that association (often with benefits to the design of the domain model) with an entity type and several binary associations: The new entity type represents, as first-class entities in the domain model, the links of the non-binary association that it replaces.

Therefore, following the fourth guiding-principle of Section 1.2.2, I considered that the eventual benefits of supporting n-ary associations in DML were not worthy of the effort necessary to implement them.

5.6.1 Syntax of Association Declarations

An *association declaration*, which adds an association to a domain specification, is a top-level construct in the DML language—that is, it is a construct that stands by itself, rather than being part of, or having to appear subordinated to another construct.

This syntax for associations contrasts with how other approaches to the problem of making associations explicit in the programming language propose to represent associations. Many of such approaches propose to represent associations with new constructs that programmers should use in each of the associated classes. This solution, however, splits the information about the association between the two classes, making both the understanding and the maintenance of a domain model more difficult. Thus, I argue that an association declaration should be a single construct, as it is in the DML language. In Listing 5.6 on the next page, I show the grammar rules that specify the syntax of an association declaration. This set of rules concludes the syntactic grammar of the DML language.

The keyword `association` introduces an association declaration and is followed by the name of the association, an identifier. As we shall see in the following section, the DML compiler uses this identifier to name a static field in each of the classes that participate in the association. Therefore, this name must be unique throughout the inheritance hierarchy of each of those classes. To avoid name clashes, programmers should adopt a naming convention that distinguishes the names of associations from the names of other members of a class (e.g., inner classes).

In a relationship among several entities, each of the entities plays a role in the relationship. Thus, the DML language uses role declarations to identify the entity types that participate in an association. As in DML all the associations are binary, an association declaration has exactly two role declarations.

Each role declaration specifies the type of the entities that play the declared role in the association, the name of the role, and the multiplicity of the role. Yet, both the name and the multiplicity of the role are optional. If the name of a role is not specified, then the association is not traversable in that direction—that is, it is not possible to reach the entities that play that role from the entities in the other end of the association. I shall discuss this further in the following section, where I present the semantics of an association declaration.

If the multiplicity of a role is not specified, however, a default multiplicity of `0..1`

AssociationDeclaration:
`association` *Identifier RoleDeclarations*

RoleDeclarations:
`{` *RoleDeclaration RoleDeclaration* `}`

RoleDeclaration:
EntityTypeName `playsRole` *Identifier*^{opt} *RoleBody*

RoleBody:
`;`
`{` *MultiplicityOption* `}`

MultiplicityOption:
`multiplicity` *MultiplicityValues* `;`

MultiplicityValues:
MultiplicityRange
MultiplicityValues `,` *MultiplicityRange*

MultiplicityRange:
MultiplicityUpperBound
MultiplicityLowerBound `..` *MultiplicityUpperBound*

MultiplicityUpperBound:
`*`
DecimalNumeral

MultiplicityLowerBound:
DecimalNumeral

Listing 5.6: Grammar rules for the syntax of an association declaration. Each association declaration has exactly two role declarations, identifying the two entity types that participate in the relationship. The nonterminal symbol *DecimalNumeral* is defined in the Java lexical grammar [Gosling et al., 2005] either as the digit `0` or as a sequence of digits that start with a non-zero digit.

```
association AccountOwnership {
  Client playsRole owner {
    multiplicity 1;  // it is equivalent to 1..1
  }

  ClientAccount playsRole account {
    multiplicity 1..*;
  }
}

association AccountGroup {
  CheckingAccount playsRole checking {
    multiplicity 1;
  }

  SavingsAccount playsRole savings {
    multiplicity *;  // it is equivalent to 0..*
  }
}
```

Listing 5.7: Examples of association declarations in DML. Both associations are bidirectional one-to-many associations. Note that the role names and the multiplicities match those specified in the class diagram shown in Figure 2.4 on page 24.

is assumed. Moreover, the multiplicity range `*` is a shorthand for `0..*`. Finally, a multiplicity range of the form `N`, for any positive integer *N*, is a shorthand for `N..N`. Thus, every role has a multiplicity value, either implicitly or explicitly declared, which is a sequence of one or more ranges with a lower bound and an upper bound.

Note that the syntax for an association declaration could be made simpler. For instance, we could eliminate the keywords `playsRole` and `multiplicity`, as well as some of the curly braces. The reason for having this syntax, however, is that I designed it with extensibility in mind. For example, I foresee the need to add other options to a role declaration. Yet, given the current syntax for the multiplicity option, the syntactic changes to add other options would be minimal.

To conclude this section I show, in Listing 5.7, two association declarations that represent two of the associations from the banking domain.

5.6.2 Semantics of Association Declarations

Like before, when I specified the semantics of an entity type declaration, I specify the semantics of an association declaration by prescribing the minimal Java interface that a DML compiler must generate for each association declaration.

In this case, however, the DML compiler does not need to generate any new classes to

implement an association declaration. Rather, it must generate new methods for each of the entity types that play a role in the association.

The semantics that I specify here tries to respect the pragmatics of object-oriented programming. Object-oriented programmers do not, as a common practice, implement a binary association or the association's instances—the links—as first-class objects in the program. Instead, the object-oriented common practice is to implement a binary association between two classes *A* and *B*, by adding to each of the classes *A* and *B* a reference to the elements of the class in the other side of the association: In the class *A* we add a reference to elements of *B*; in the class *B* we add a reference to elements of *A*. These references, however, are not typically accessible by the classes' clients. As we have seen in Section 3.3.2, the usual approach is to provide methods that access these references.

Thus, because programmers should use the public methods, rather than the private references, the semantics that I specify here prescribes only which methods must a DML compiler generate. It does not specify how the compiler should implement those methods. It does not specify either, how the generated code should implement the association's links, whether with references in each of the classes, or with any other solution.

To simplify the presentation below, given an association declaration with two role declarations, I use the term **opposite type** of a role declaration to refer to the entity type of the other role declaration. For example, given the following association declaration

```
association Rel {
  A playsRole role1;
  B playsRole role2;
}
```

I say that the opposite type of the first role declaration is `B`, and that the opposite type of the second role declaration is `A`.

A compiler for the DML language must generate at most two sets of related methods for an association declaration—one set of methods for each of the role declarations that have a role name specified. If a role declaration has no name, then the DML compiler should not generate any methods for that role declaration. The methods generated from a role declaration belong to the opposite type of that role declaration. They allow the traversal from an instance of the opposite type to one or more instances of the role declaration's type.

Therefore, an association where both role declarations have a name is a bidirectional association, whereas if only one role declaration has a name, the association is unidirectional. An association declaration where none of the role declaration have a name is meaningless for a domain specification, because no code will be generated from it.

The signature of the methods that a DML compiler must generate for a role declaration depends only on the properties of that role declaration:

- The role's multiplicity determines the set of methods to generate. Although the multiplicity option may have many different values, the DML compiler separates the roles into two disjoint classes: (1) the roles that have a multiplicity upper-bound of one; and (2) the roles that have a multiplicity upper-bound greater than one.[2] The multiplicity upper-bound of a role declaration is the maximum of the upper-bounds of all the multiplicity ranges (there is at least one) in the role declaration. The upper-bound of a multiplicity range of the form `M..N` is infinite, if `N` is the terminal symbol `*`, and the value of the integer `N`, otherwise. I shall present below, separately, the set of methods that must be generated for each of these cases.

- The role's name determines the exact name of each method. The name of the role with the first letter capitalized appears in all the methods' names, either with a prefix only, or with both a prefix and a suffix. Below, when I present the methods to generate, I show in *italic* the part of the name of each method that must be replaced by the role's name.

- The role's type determines the type of the argument, or the return type of some of the methods to generate.

Regardless of the multiplicities involved, the methods that implement an association must allow us to do each one of the following tasks: (1) create a new link between two objects, (2) remove an existing link between two objects, and (3) traverse from one object in one end of a link to the object on the other end. The specific signature of the methods that allow us to do this, however, depends on the multiplicity of each role. I shall present each case separately.

5.6.2.1 Roles with a multiplicity upper-bound of one

If the multiplicity of a role declaration has an upper-bound of one, then an object of the opposite type may refer to at most one object of the role's type. This is the simplest case of an association, which is typically implemented with a getter and a setter method.

In Figure 5.4 on the next page, I show the signature of the methods that a DML compiler must generate for a role declaration with a multiplicity upper-bound of one. Only the first two methods, shown in **boldface**, are necessary to implement the association. The other two methods may be trivially implemented by using the mandatory methods. Yet, they make the class more convenient to use, and, therefore, more programmer-friendly.

[2] Roles with a multiplicity upper-bound lesser than one do not make sense.

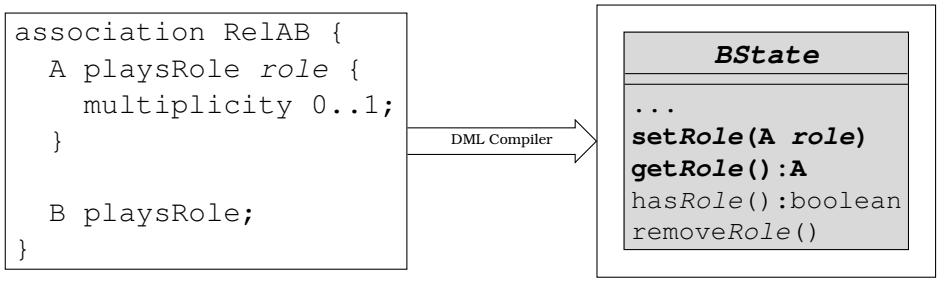

Figure 5.4: Methods that a DML compiler must generate for a role declaration with a multiplicity upper-bound of one. The part of a method's name that depends on the name of the role is shown in *italic*. The methods in **boldface** are the core methods. The other methods are optional methods.

The method `set`*`Role`* is the setter method. It allows us to create or to eliminate a link between an instance of class `B` and an instance of class `A`. To create a link, we should call the method with an instance of class `A` as argument. In this case, the method execution creates a new link between the receiver and the argument of the method call. But, because we can have at most an instance `A` linked to each instance of `B`, if there was already a link between the receiver of the method call and another instance of class `A`, the method must eliminate that link before it creates the new link. Finally, the call to the setter method with a `null` argument eliminates any current link that may exist for the receiver of the method call.

The method `get`*`Role`* is the getter method. Given an instance of the class `B`, `b`, the method call `b.getRole()` returns the instance of the class `A` that is related to `b`, if such an instance exists, or `null`, otherwise. This method is the method that allows us to traverse the association.

The method `has`*`Role`* returns true if there is a link between the receiver of the method call and an instance of class `A`. Otherwise, it returns false. Calling this method on an object `obj` is equivalent to evaluate the Java expression `obj.getRole() != null`. The method, however, may be implemented more efficiently by the DML compiler.

Finally, the method `remove`*`Role`* removes an existing link that may exist for the receiver of the method call. It is equivalent to call the setter method with a `null` argument.

5.6.2.2 Roles with a multiplicity upper-bound greater than one

The difference between this case and the previous is that the object of the opposite type may have multiple links, at the same time, with objects of the role's type. So, when we create a new link, we do not remove any existing link, as in the previous case. The existing links must be removed explicitly, by specifying which object is on the other end of the link that should be removed. Furthermore, when we traverse the association, we may reach multiple objects. Thus, the getter method, in this case, must return a collection of

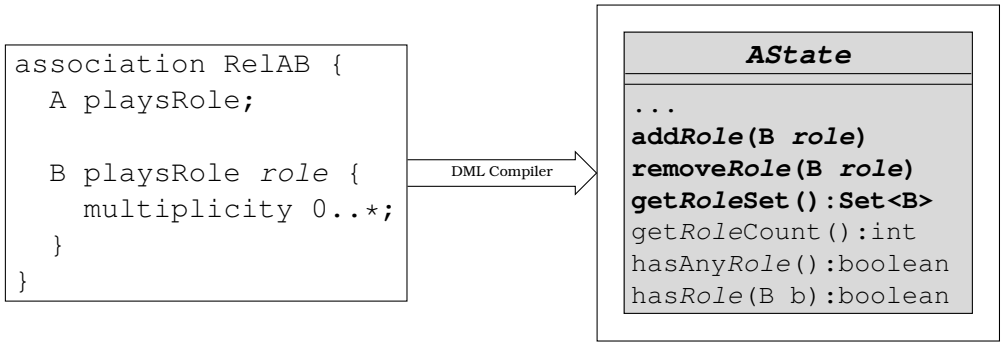

Figure 5.5: Methods that a DML compiler must generate for a role declaration with a multiplicity upper-bound greater than one. The part of a method's name that depends on the name of the role is shown in *italic*. The methods in **boldface** are the core methods. The other methods are optional methods.

objects.

In Figure 5.5, I show the signature of the methods that a DML compiler must generate for this case. Like in the previous case, there is a set of core methods, and a set of convenience methods.

The method `addRole` creates a new link between the receiver of the method and an instance of the class `B` passed as argument to the method. If the argument of the method is `null`, the method does nothing.

The method `removeRole` eliminates the link between the receiver of the method and the instance of the class `B` passed as argument to the method. If no such link exists, or if the argument is `null`, the method has no effect.

The method `getRoleSet` returns the set of instances of the class `B` that have a link with the receiver of the method. The value returned by this method is always an instance of a class that implements the `java.util.Set` interface, but it must satisfy some conditions that are specified below.

Finally, the remaining three methods may be defined by the expressions to which they are equivalent:

```
// the expression
obj.getRoleCount()
// is equivalent to the expression
obj.getRoleSet().size()

// the expression
obj.hasAnyRole()
// is equivalent to the expression
(! obj.getRoleSet().isEmpty())

// the expression
```

```
obj.hasRole(b)
// is equivalent to the expression
obj.getRoleSet().contains(b)
```

5.6.2.3 Bidirectional associations

When a new link is created for a bidirectional association, that new link must become traversable in both directions, rather than in only one, regardless of how that link is created. Likewise for the removal of a link.

A link is an instance of an association between two objects—that is, it represents that the two objects are related by the relationship that the association represents. A link, however, is typically implemented as two separate references: Each of the objects in the link keeps a reference to the other. Yet, when a link is created or eliminated both references must be updated to reflect the change.

Thus, in a bidirectional association, we should be able to create the same link by calling either one method for the object in one end of the link, or calling an equivalent method for the object in the other end. For instance, if we merge the two declarations for the association `RelAB` (see Figure 5.4 on page 117 and Figure 5.5 on the facing page), the execution of the code `a.addRole(b)` should be equivalent to the execution of the code `b.setRole(a)`.

As a final example, consider that we have the following association declaration

```
association Rel {
  A playsRole a { multiplicity 0..1; }
  B playsRole b { multiplicity 0..1; }
}
```

Moreover, consider that we have two instances of the class `A`, `a1` and `a2`, and two instances of the class `B`, `b1` and `b2`. Between `a1` and `b1` there is a link, and another link exists between `a2` and `b2`. Thus, `a1.getB()` returns `b1`, `b1.getA()` returns `a1`, `a2.getB()` returns `b2`, and `b2.getA()` returns `a2`.

Consider now that we execute either `a1.setB(b2)`, or the equivalent `b2.setA(a1)`. Either of the calls creates a new link between `a1` and `b2`, but, as the multiplicity upper-bound of both roles is one, the previous links must be eliminated. Thus, after the execution of either of the calls, we have that `a1.getB()` returns `b2`, `b1.getA()` returns `null`, `a2.getB()` returns `null`, and `b2.getA()` returns `a1`.

5.6.2.4 Sets returned by the method `get`*`Role`*`Set`

The specification for the method `get`*`Role`*`Set`, presented above, does not specify the exact nature of the value returned by the method—for example, whether the value returned is an immutable set or a mutable set. Yet, it is important to specify these details, so that programmers know what they can and cannot do with the result returned.

In the DML language, the value returned by a call to the method `get`*`Role`*`Set` is a mutable set that is backed up by the association and the receiver of the method call. This means not only that programmers may add and remove elements from the set, but also that those changes correspond, in fact, to the creation and elimination of association links. Furthermore, if a new link is added to or removed from the object that backs up the set, the set reflects that change.

The advantage of having this specification is that we may use all the methods available in the `java.util.Set` interface to manipulate the links of an object. For instance, we may remove all the links of an object, `o`, with the code `o.getRoleSet().clear()`. Or, probably more useful, we may iterate over the set and remove the elements that satisfy some condition.

Finally, a consequence of this specification is that the method call `o1.addRole(o2)` is equivalent to `o1.getRoleSet().add(o2)`. Likewise for the method that removes a link. Thus, the method `get`*`Role`*`Set` is sufficient to implement a role declaration with a multiplicity upper-bound greater than one.

5.6.2.5 Association objects and their listeners

Having different ways to create and to remove a link makes the domain model more programmer-friendly, because programmers may choose what is more convenient for them in each situation. Yet, all these equivalent approaches make the domain model more complex, and, eventually, more confusing, also. In particular, having various entry points for the same functionality raises the question of which method should we override if we want to customize that functionality.

Imagine that we want to execute some code whenever a new link for the association `RelAB` is created between an instance of the class `A` and an instance of the class `B`. Should we override the method `set`*`Role`* in the class `B`? That would work for the links created by that method. But a link may be created by calling the method `add`*`Role`* in the class `A`, also, and we do not know whether that method calls the method `set`*`Role`* or not. Yet, if we override the method `add`*`Role`* instead, we have similar problems. Even if we override both methods, we are not sure to catch all the link creations because links may be created when we add objects to a set returned by the method `get`*`Role`*`Set`.

```
interface Association<C1,C2> {
  void add(C1 o1, C2 o2);
  void remove(C1 o1, C2 o2);
  Association<C2,C1> getInverse();
  void addListener(AssocListener<C1,C2> listener);
  void removeListener(AssocListener<C1,C2> listener);
}

interface AssocListener<C1,C2> {
  void beforeAdd(Association<C1,C2> assoc, C1 o1, C2 o2);
  void afterAdd(Association<C1,C2> assoc, C1 o1, C2 o2);
  void beforeRemove(Association<C1,C2> assoc, C1 o1, C2 o2);
  void afterRemove(Association<C1,C2> assoc, C1 o1, C2 o2);
}
```

Listing 5.8: The Java generic interfaces `Association` and `AssocListener`. The DML compiler uses objects of the `Association` type to implement an association in Java. Programmers may customize the association operations by adding listeners to an association object.

To solve this problem, I propose to add yet another way to create and to remove links from an association: A common point in the code that must be executed whenever a link is created or removed, regardless of how that is done.

The key idea is to have an object that represents an association. The basic operations of that object are a method to add a new pair of objects to the association, and a method to remove a pair of objects from the association. The first method creates a new link, the second removes an existing link. These methods must be called to create or to remove a link. In fact, all the methods described previously that create or remove links may call the add or the remove operations of this new object. The semantics must be the same.

The association object gives us a central point of execution for the operations that change an association. Now, we need some mechanism to specialize those operations. Because each association is represented by an object, rather than by a class, we cannot use standard class inheritance and method overriding to do that specialization. Instead, we may use listeners to do it. In Listing 5.8, I show both the interface `Association` and the interface `AssocListener` that will allow us to specialize the creation and the removal of a link.

The method `addListener` adds a new listener to an association object, whereas the method `removeListener` removes a listener. Even though these methods may be used at runtime to change the set of listeners for an association object dynamically, the common usage is to add one or more listeners to an association object at class-load-time and use that set of listeners throughout the entire program.

When an association object has some listeners registered, the methods `add` and

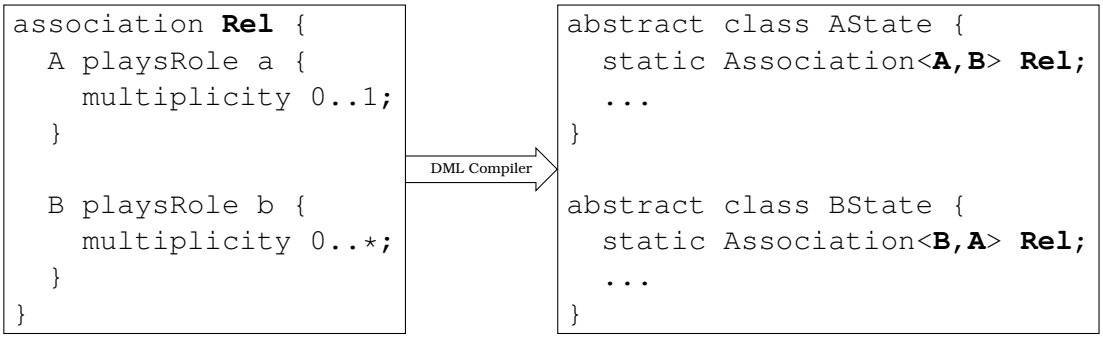

Figure 5.6: Static fields that a DML compiler must generate to hold the `Association` objects. The name of the static field in each class is the name of the association, as given in the association declaration. The association object in each class is the inverse of the association object in the other class.

`remove` call the appropriate methods of the registered listeners, following the same order by which the listeners were added to the association object. The method `add` first calls the method `beforeAdd` of each listener, then creates the link, and, finally, calls the method `afterAdd` of each listener. Mutatis mutandis for the method `remove`. Note that a listener may cancel the creation (or the removal) of a link, if its `beforeAdd` (or `beforeRemove`) method throws an exception.

Using custom association listeners, it is easy to specialize an association. But, before we can do that, we need to know how to access the objects that represent associations in a domain model. My proposal is to make them accessible through static fields in the classes that participate in the association, as depicted in Figure 5.6.

With this final piece in place, we may now specialize the creation of a link, as intended. In Listing 5.9 on the next page, I show a sketch of the code that we may add to the behavior class `A` to accomplish that.

5.7 Implementation of a Domain Specification

The specification of the DML language prescribes in detail the interface of the classes that a conforming DML compiler must generate; it does not, however, dictate how that interface should be implemented, even though it gives, now and then, some hints about possible implementation strategies.

Therefore, different compilers for the DML language may generate different source code to implement a domain specification: Either because they must generate domain models with different properties, or simply because they use different implementation strategies.

In fact, the simplicity of the DML language is a conscious design decision to promote experimentation with different implementation strategies, which may affect significantly