10

Mapping Models to Code

The major difference between a thing that might go wrong and a thing that cannot possibly go wrong is that when a thing that cannot possibly go wrong goes wrong it usually turns out to be impossible to get at or repair.

—Douglas Adams, in *Mostly Harmless*

If the design pattern selection and the specification of class interfaces were done carefully, most design issues should now be resolved. We could implement a system that realizes the use cases specified during requirements elicitation and system design. However, as developers start putting together the individual subsystems developed in this way, they are confronted with many integration problems. Different developers have probably handled contract violations differently. Undocumented parameters may have been added to the API to address a requirement change. Additional attributes have possibly been added to the object model, but are not handled by the persistent management system, possibly because of a miscommunication. As the delivery pressure increases, addressing these problems results in additional improvised code changes and workarounds that eventually yield to the degradation of the system. The resulting code would have little resemblance to our original design and would be difficult to understand.

In this chapter, we describe a selection of transformations to illustrate a disciplined approach to implementation to avoid such a system degradation. These include

- optimizing the class model
- mapping associations to collections
- mapping operation contracts to exceptions
- mapping the class model to a storage schema.

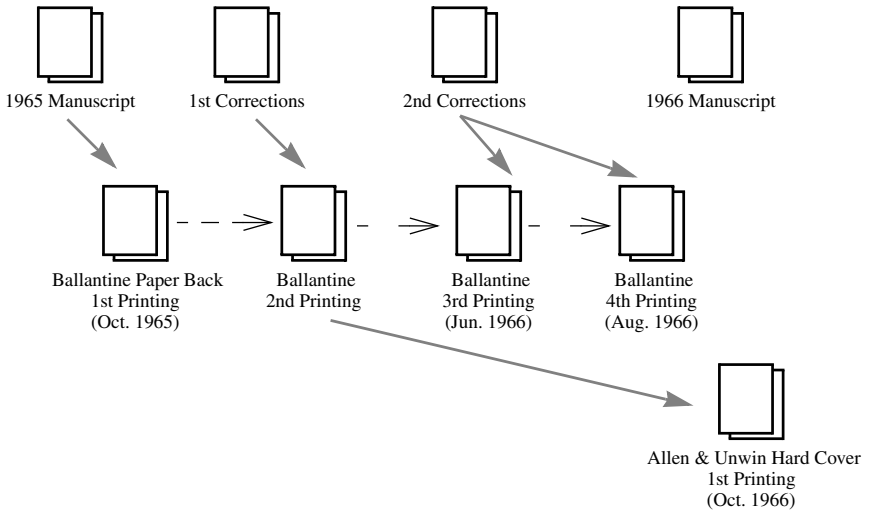
We use Java and Java-based technologies in this chapter. The techniques we describe, however, are also applicable to other object-oriented programming languages.

10.1 Introduction: A Book Example

The Lord of the Rings by J.R.R. Tolkien was first published in the U.K. in 1954 and in the U.S. in 1955.[a] Tolkien, a professor of English and a distinguished linguist, invented several languages and carefully constructed many imaginary names of places and characters for the universe of *The Lord of the Rings*. However, when the book was first published, it contained many typesetting and printing errors and well-intended corrections. For example, to Tolkien's horror, occurrences of *dwarves* and *elven* were systematically replaced by *dwarfs* and *elfin.* Then, in 1965, Ace Books published an unauthorized edition, believing that the copyright on the book did not apply in the United States. Tolkien started to work on a revision of the text so that a new authorized edition would compete against the Ace Books edition. Ballantine Books published the revised manuscript as a paperback in the U.S. in October 1965. Tolkien produced a first set of corrections that were implemented in the second printing of the paperback. Soon after, he produced a second set of corrections, which were implemented in the third and fourth printing of the paperback. These corrections were not always inserted correctly, resulting in more errors in the text.

In the U.K., Allen & Unwin published the revised manuscript as a hardcover edition in October 1966. However, the revised manuscript for the appendix of *The Lord of the Rings*, which Ballantine Books used to set the paperback edition, was lost and could not be used by Allen & Unwin. Instead, Allen & Unwin used an early copy of the Ballantine edition to set the appendix. Unfortunately, the copy did not include the second set of corrections, and, worse, many more errors were introduced during typesetting.

During the summer of 1966, Tolkien further revised the text. In June 1966, he was informed that this latest set of revisions was too late to be included into the Allen & Unwin edition. Consequently, by the end of 1966, there existed several widely different and inconsistent versions of *The Lord of the Rings*, none of which completely reflected Tolkien's intention.



a. Source: "Note on The Text" by Douglas A. Anderson, in [Tolkien, 1995].

The publication of a book, such as *The Lord of the Rings*, requires a number of transformation steps from manuscript to the final printed book. The author first produces a typed manuscript and sends it to a publisher. The manuscript is then revised and copy edited before being finally typeset. The typesetting process requires the retyping of the manuscript, resulting in mistakes, such as misspellings and well-intended corrections. Finally, during the printing process, other mistakes occur, such as wrong ordering or orientation of printed pages.

Although many steps of the publishing process are now automated, mistakes still creep in, introduced by the author, the publisher, or the printer, resulting in a series of corrections that can introduce new errors. Although the processes involved are relatively simple (retyping a manuscript is intellectually not very challenging), the sheer volume and repetitiveness of the work makes it difficult to accomplish without errors (the six books of *The Lord of the Rings* total more than one thousand pages).

Working on an object design model similarly involves many transformations that are error prone. For example, developers perform local transformations to the object model to improve its modularity and performance. Developers transform the associations of the object model into collections of object references, because programming languages do not support the concept of association. Many programming languages also do not support contracts, so developers need to transform the contract specification of operations into code for detecting and handling contract violations. Developers revise the interface specification of an object to accommodate new requirements from the client. These transformations are not as intellectually challenging as other activities in the development process, such as analysis or system design, but they have a repetitive and mechanical aspect and are made difficult by the scale of the work involved. Moreover, they are accomplished mostly manually, as they require some amount of human judgement.

In each of these transformations, small errors usually creep in, resulting in bugs and test failures. We now focus on a set of techniques and example transformations to reduce the number of such errors.

10.2 An Overview of Mapping

A **transformation** aims at improving one aspect of the model (e.g., its modularity) while preserving all of its other properties (e.g., its functionality). Hence, a transformation is usually localized, affects a small number of classes, attributes, and operations, and is executed in a series of small steps. These transformations occur during numerous object design and implementation activities. We focus in detail on the following activities:

- *Optimization* (Section 10.4.1). This activity addresses the performance requirements of the system model. This includes reducing the multiplicities of associations to speed up queries, adding redundant associations for efficiency, and adding derived attributes to improve the access time to objects.

- *Realizing associations* (Section 10.4.2). During this activity, we map associations to source code constructs, such as references and collections of references.
- *Mapping contracts to exceptions* (Section 10.4.3). During this activity, we describe the behavior of operations when contracts are broken. This includes raising exceptions when violations are detected and handling exceptions in higher level layers of the system.
- *Mapping class models to a storage schema* (Section 10.4.4). During system design, we selected a persistent storage strategy, such as a database management system, a set of flat files, or a combination of both. During this activity, we map the class model to a storage schema, such as a relational database schema.

10.3 Mapping Concepts

We distinguish four types of transformations (Figure 10-1):

- *Model transformations* operate on object models (Section 10.3.1). An example is the conversion of a simple attribute (e.g., an address represented as a string) to a class (e.g., a class with street address, zip code, city, state, and country attributes).
- *Refactorings* are transformations that operate on source code (Section 10.3.2). They are similar to object model transformations in that they improve a single aspect of the system without changing its functionality. They differ in that they manipulate the source code.
- *Forward engineering* produces a source code template that corresponds to an object model (Section 10.3.3). Many modeling constructs, such as attribute and association specifications, can be mechanically mapped to source code constructs supported by the selected programming language (e.g., class and field declarations in Java), while the bodies and additional private methods are added by developers.
- *Reverse engineering* produces a model that corresponds to source code (Section 10.3.4). This transformation is used when the design of the system has been lost and must be recovered from the source code. Although several CASE tools support

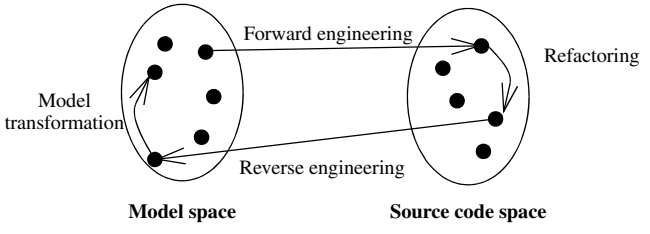


Figure 10-1 The four types of transformations described in this chapter: model transformations, refactorings, forward engineering, and reverse engineering.

reverse engineering, much human interaction is involved for recreating an accurate model, as the code does not include all information needed to recover the model unambiguously.

10.3.1 Model Transformation

A **model transformation** is applied to an object model and results in another object model [Blaha & Premerlani, 1998]. The purpose of object model transformation is to simplify or optimize the original model, bringing it into closer compliance with all requirements in the specification. A transformation may add, remove, or rename classes, operations, associations, or attributes. A transformation can also add information to the model or remove information from it.

In Chapter 5, *Analysis*, we used transformations to organize objects into inheritance hierarchies and eliminate redundancy from the analysis model. For example, the transformation in Figure 10-2 takes a class model with a number of classes that contain the same attribute and removes the redundancy. The `Player`, `Advertiser`, and `LeagueOwner` in `ARENA` all have an `email` address attribute. We create a superclass `User` and move the `email` attribute to the superclass.

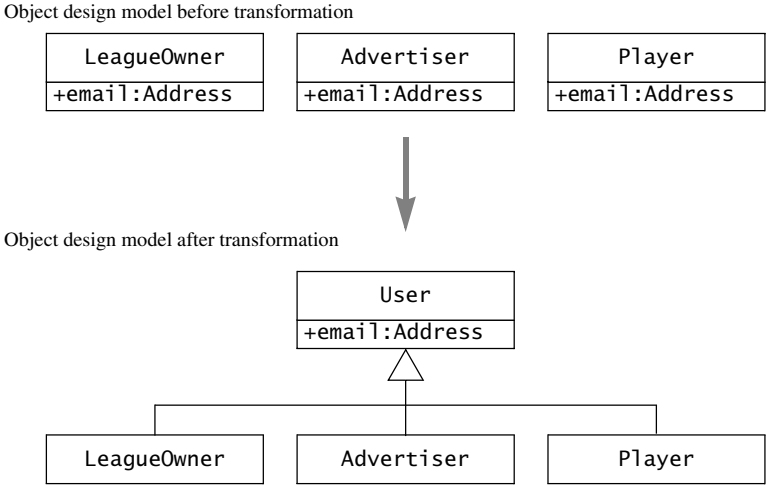


Figure 10-2 An example of an object model transformation. A redundant attribute can be eliminated by creating a superclass.

In principle, the development process can be thought as a series of model transformations, starting with the analysis model and ending with the object design model, adding solution domain details along the way. Although applying a model transformation is a fairly mechanical activity, identifying which transformation to apply to which set of classes requires judgement and experience.

10.3.2 Refactoring

A **refactoring** is a transformation of the source code that improves its readability or modifiability without changing the behavior of the system [Fowler, 2000]. Refactoring aims at improving the design of a working system by focusing on a specific field or method of a class. To ensure that the refactoring does not change the behavior of the system, the refactoring is done in small incremental steps that are interleaved with tests. The existence of a test driver for each class allows developers to confidently change the code and encourages them to change the interface of the class as little as possible during the refactoring.

For example, the object model transformation of Figure 10-2 corresponds to a sequence of three refactorings. The first one, Pull Up Field, moves the `email` field from the subclasses to the superclass `User`. The second one, Pull Up Constructor Body, moves the initialization code from the subclasses to the superclass. The third and final one, Pull Up Method, moves the methods manipulating the `email` field from the subclasses to the superclass. Let's examine these three refactorings in detail.

Pull Up Field relocates the `email` field using the following steps (Figure 10-3):

1. Inspect `Player`, `LeagueOwner`, and `Advertiser` to ensure that the `email` field is equivalent. Rename equivalent fields to `email` if necessary.
2. Create public class `User`.
3. Set parent of `Player`, `LeagueOwner`, and `Advertiser` to `User`.
4. Add a protected field `email` to class `User`.
5. Remove fields `email` from `Player`, `LeagueOwner`, and `Advertiser`.
6. Compile and test.

Before refactoring

```
public class Player {
    private String email;
    //...
}
public class LeagueOwner {
    private String eMail;
    //...
}
public class Advertiser {
    private String email_address;
    //...
}
```

After refactoring

```
public class User {
    protected String email;
}
public class Player extends User {
    //...
}
public class LeagueOwner extends User
{
    //...
}
public class Advertiser extends User {
    //...
}
```

Figure 10-3 Applying the *Pull Up Field* refactoring.

Then, we apply the *Pull Up Constructor Body* refactoring to move the initialization code for `email` using the following steps (Figure 10-4):

1. Add the constructor `User(Address email)` to class `User`.
2. Assign the field `email` in the constructor with the value passed in the parameter.
3. Add the call `super(email)` to the `Player` class constructor.
4. Compile and test.
5. Repeat steps 1–4 for the classes `LeagueOwner` and `Advertiser`.

Before refactoring

```
public class User {
    private String email;
}

public class Player extends User {
    public Player(String email) {
        this.email = email;
        //...
    }
}
public class LeagueOwner extends User
{
    public LeagueOwner(String email) {
        this.email = email;
        //...
    }
}
public class Advertiser extends User {
    public Advertiser(String email) {
        this.email = email;
    //...
    }
}
```

After refactoring

```
public class User {
    public User(String email) {
        this.email = email;
    }
}
public class Player extends User {
    public Player(String email) {
        super(email);
    //...
    }
}
public class LeagueOwner extends User
{
    public LeagueOwner(String email) {
        super(email);
    //...
    }
}
public class Advertiser extends User {
    public Advertiser(String email) {
        super(email);
    //...
    }
}
```

Figure 10-4 Applying the *Pull Up Constructor Body* refactoring.

At this point, the field `email` and its corresponding initialization code are in the `User` class. Now, we examine if methods using the `email` field can be moved from the subclasses to the `User` class. To achieve this, we apply the *Pull Up Method* refactoring:

1. Examine the methods of `Player` that use the `email` field. Note that `Player.notify()` uses `email` and that it does not use any fields or operations that are specific to `Player`.
2. Copy the `Player.notify()` method to the `User` class and recompile.
3. Remove the `Player.notify()` method.

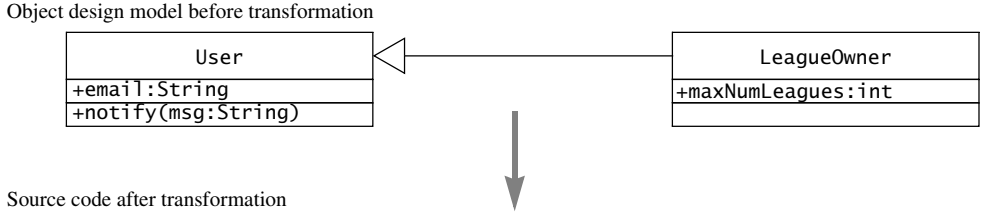
4. Compile and test.
5. Repeat for `LeagueOwner` and `Advertiser`.

Applying these three refactorings effectively transforms the `ARENA` source code in the same way the object model transformation of Figure 10-2 transformed the `ARENA` object design model. Note that the refactorings include many more steps than its corresponding object model transformation and interleave testing with changes. This is because the source code includes many more details, so it provides many more opportunities for introducing errors. In the next section, we discuss general principles for avoiding transformation errors.

10.3.3 Forward Engineering

Forward engineering is applied to a set of model elements and results in a set of corresponding source code statements, such as a class declaration, a Java expression, or a database schema. The purpose of forward engineering is to maintain a strong correspondence between the object design model and the code, and to reduce the number of errors introduced during implementation, thereby decreasing implementation effort.

For example, Figure 10-5 depicts a particular forward engineering transformation applied to the classes `User` and `LeagueOwner`. First, each UML class is mapped to a Java class. Next, the



Source code after transformation

```
public class User {
    private String email;
    public String getEmail() {
        return email;
    }
    public void setEmail(String
value){
        email = value;
    }
    public void notify(String msg) {
        // ....
    }
    /* Other methods omitted */
}
```

```
public class LeagueOwner extends User
{
    private int maxNumLeagues;
    public int getMaxNumLeagues() {
        return maxNumLeagues;
    }
    public void setMaxNumLeagues
                        (int value) {
        maxNumLeagues = value;
    }
    /* Other methods omitted */
}
```

Figure 10-5 Realization of the `User` and `LeagueOwner` classes (UML class diagram and Java excerpts). In this transformation, the public visibility of `email` and `maxNumLeagues` denotes that the methods for getting and setting their values are public. The actual fields representing these attributes are private.

UML generalization relationship is mapped to an `extends` statement in the `LeagueOwner` class. Finally, each attribute in the UML model is mapped to a private field in the Java classes and to two public methods for setting and getting the value of the field. Developers can then refine the result of the transformation with additional behavior, for example, to check that the new value of `maxNumLeagues` is a positive integer.

Note that, except for the names of the attributes and methods, the code resulting from this transformation is always the same. This makes it easier for developers to recognize transformations in the source code, which encourages them to comply with naming conventions. Moreover, since developers use one consistent approach for realizing classes, they introduce fewer errors.

10.3.4 Reverse Engineering

Reverse engineering is applied to a set of source code elements and results in a set of model elements. The purpose of this type of transformation is to recreate the model for an existing system, either because the model was lost or never created, or because it became out of sync with the source code. Reverse engineering is essentially an inverse transformation of forward engineering. Reverse engineering creates a UML class for each class declaration statement, adds an attribute for each field, and adds an operation for each method. However, because forward engineering can lose information (e.g., associations are turned into collections of references), reverse engineering does not necessarily recreate the same model. Although many CASE tools support reverse engineering, CASE tools provide, at best, an approximation that the developer can use to rediscover the original model.

10.3.5 Transformation Principles

A transformation aims at improving the design of the system with respect to some criterion. We discussed four types of transformations so far: model transformations, refactorings, forward engineering, and reverse engineering. A model transformation improves the compliance of the object design model with a design goal. A refactoring improves the readability or the modifiability of the source code. Forward engineering improves the consistency of the source code with respect to the object design model. Reverse engineering tries to discover the design behind the source code.

However, by trying to improve one aspect of the system, the developer runs the risk of introducing errors that will be difficult to detect and repair. To avoid introducing new errors, all transformations should follow these principles:

- *Each transformation must address a single criteria*. A transformation should improve the system with respect to only one design goal. One transformation can aim to improve response time. Another transformation can aim to improve coherence. However, a transformation should not optimize multiple criteria. If you find yourself

trying to deal with several criteria at once, you most likely introduce errors by making the source code too complex.

- *Each transformation must be local*. A transformation should change only a few methods or a few classes at once. Transformations often target the implementation of a method, in which case the callers are not affected. If a transformation changes an interface (e.g., adding a parameter to a method), then the client classes should be changed one at the time (e.g., the older method should be kept around for background compatibility testing). If you find yourself changing many subsystems at once, you are performing an architectural change, not an object model transformation.
- *Each transformation must be applied in isolation to other changes*. To further localize changes, transformations should be applied one at the time. If you are improving the performance of a method, you should not add new functionality. If you are adding new functionality, you should not optimize existing code. This enables you to focus on a limited set of issues and reduces the opportunities for errors.
- *Each transformation must be followed by a validation step*. Even though transformations have a mechanical aspect, they are applied by humans. After completing a transformation and before initiating the next one, validate the changes. If you applied an object model transformation, update the sequence diagrams in which the classes under consideration are involved. Review the use cases related to the sequence diagrams to ensure that the correct functionality is provided. If you applied a refactoring, run the test cases relevant to the classes under consideration. If you added new control statements or dealt with new boundary cases, write new tests to exercise the new source code. It is always easier to find and repair a bug shortly after it was introduced than later.

10.4 Mapping Activities

In this section, we present transformations that occur frequently to illustrate the principles we described in the previous section. We focus on transformations during the following activities:

- Optimizing the Object Design Model (Section 10.4.1)
- Mapping Associations to Collections (Section 10.4.2)
- Mapping Contracts to Exceptions (Section 10.4.3)
- Mapping Object Models to a Persistent Storage Schema (Section 10.4.4).

10.4.1 Optimizing the Object Design Model

The direct translation of an analysis model into source code is often inefficient. The analysis model focuses on the functionality of the system and does not take into account system design decisions. During object design, we transform the object model to meet the design goals identified during system design, such as minimization of response time, execution time, or memory resources. For example, in the case of a Web browser, it might be clearer to represent

HTML documents as aggregates of text and images. However, if we decided during system design to display documents as they are retrieved, we may introduce a proxy object to represent placeholders for images that have not yet been retrieved.

In this section, we describe four simple but common optimizations: adding associations to optimize access paths, collapsing objects into attributes, delaying expensive computations, and caching the results of expensive computations.

When applying optimizations, developers must strike a balance between efficiency and clarity. Optimizations increase the efficiency of the system but also the complexity of the models, making it more difficult to understand the system.

Optimizing access paths

Common sources of inefficiency are the repeated traversal of multiple associations, the traversal of associations with "many" multiplicity, and the misplacement of attributes [Rumbaugh et al., 1991].

Repeated association traversals. To identify inefficient access paths, you should identify operations that are invoked often and examine, with the help of a sequence diagram, the subset of these operations that requires multiple association traversal. Frequent operations should not require many traversals, but should have a direct connection between the querying object and the queried object. If that direct connection is missing, you should add an association between these two objects. In interface and reengineering projects, estimates for the frequency of access paths can be derived from the legacy system. In greenfield engineering projects, the frequency of access paths is more difficult to estimate. In this case, redundant associations should not be added before a dynamic analysis of the full system—for example, during system testing—has determined which associations participate in performance bottlenecks.

"Many" associations. For associations with "many" multiplicity, you should try to decrease the search time by reducing the "many" to "one." This can be done with a qualified association (Section 2.4.2). If it is not possible to reduce the multiplicity of the association, you should consider ordering or indexing the objects on the "many" side to decrease access time.

Misplaced attributes. Another source of inefficient system performance is excessive modeling. During analysis many classes are identified that turn out to have no interesting behavior. If most attributes are only involved in `set()` and `get()` operations, you should reconsider folding these attributes into the calling class. After folding several attributes, some classes may not be needed anymore and can simply removed from the model.

The systematic examination of the object model using the above questions should lead to a model with selected redundant associations, with fewer inefficient many-to-many associations, and with fewer classes.

Collapsing objects: Turning objects into attributes

After the object model is restructured and optimized a couple of times, some of its classes may have few attributes or behaviors left. Such classes, when associated only with one other class, can be collapsed into an attribute, thus reducing the overall complexity of the model.

Consider, for example, a model that includes `Persons` identified by a `SocialSecurity` object. During analysis, two classes may have been identified. Each `Person` is associated with a `SocialSecurity` class, which stores a unique social security number identifying the `Person`. Now, assume that the use cases do not require any behavior for the `SocialSecurity` object and that no other classes have associations with the `SocialSecurity` class. In this case, the `SocialSecurity` class should be collapsed into an attribute of `Person` (see Figure 10-6).

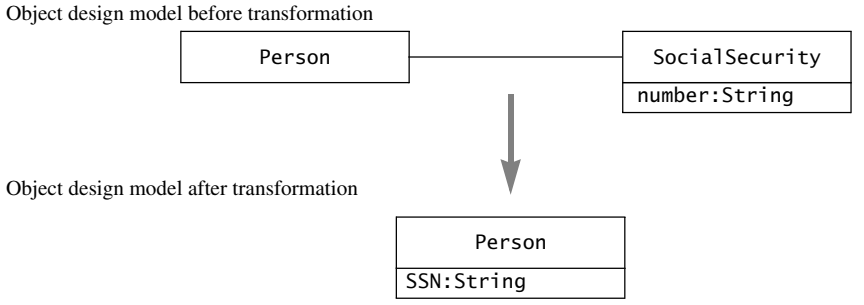


Figure 10-6 Collapsing an object without interesting behavior into an attribute (UML class diagram).

The decision of collapsing classes is not always obvious. In the case of a social security system, the `SocialSecurity` class may have much more behavior, such as specialized routines for generating new numbers based on birth dates and the location of the original application. In general, developers should delay collapsing decisions until the beginning of the implementation, when responsibilities for each class are clear. Often, this occurs after substantial coding has occurred, in which case it may be necessary to refactor the code.

The refactoring equivalent to the model transformation of Figure 10-6 is *Inline Class* refactoring [Fowler, 2000]:

1. Declare the public fields and methods of the source class (e.g., `SocialSecurity`) in the absorbing class (e.g., `Person`).
2. Change all references to the source class to the absorbing class.
3. Change the name of the source class to another name, so that the compiler catches any dangling references.
4. Compile and test.
5. Delete the source class.

Delaying expensive computations

Often, specific objects are expensive to create. However, their creation can often be delayed until their actual content is needed. For example, consider an object representing an image stored as a file (e.g., an `ARENA AdvertisementBanner`). Loading all the pixels that constitute the image from the file is expensive. However, the image data need not be loaded until the image is displayed. We can realize such an optimization using a **Proxy design pattern** [Gamma et al., 1994]. An `ImageProxy` object takes the place of the `Image` and provides the same interface as the `Image` object (Figure 10-7). Simple operations such as `width()` and `height()` are handled by `ImageProxy`. When `Image` needs to be drawn, however, `ImageProxy` loads the data from disk and creates a `RealImage` object. If the client does not invokes the `paint()` operation, the `RealImage` object is not created, thus saving substantial computation time. The calling classes only access the `ImageProxy` and the `RealImage` through the `Image` interface.

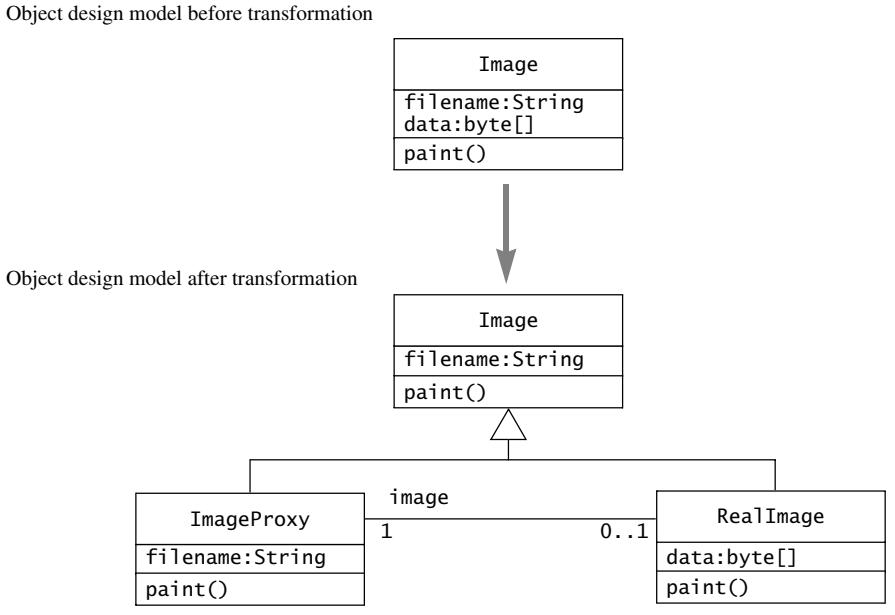


Figure 10-7 Delaying expensive computations to transform the object design model using a Proxy design pattern (UML class diagram).

Caching the result of expensive computations

Some methods are called many times, but their results are based on values that do not change or change only infrequently. Reducing the number of computations required by these methods substantially improve overall response time. In such cases, the result of the computation should be cached as a private attribute. Consider, for example, the

LeagueBoundary.getStatistics() operation, which displays the statistics relevant to all Players and Tournaments in a League. These statistics change only when a Match is completed, so it is not necessary to recompute the statistics every time a User wishes to see them. Instead, the statistics for a League can be cached in a temporary data structure, which is invalidated the next time a Match is completed. Note that this approach includes a time-space trade-off: we improve the average response time for the getStatistics() operation, but we consume memory space by storing redundant information.

10.4.2 Mapping Associations to Collections

Associations are UML concepts that denote collections of bidirectional links between two or more objects. Object-oriented programming languages, however, do not provide the concept of association. Instead, they provide references, in which one object stores a handle to another object, and collections, in which references to several objects can be stored and possibly ordered. References are unidirectional and take place between two objects. During object design, we realize associations in terms of references, taking into account the multiplicity of the associations and their direction.

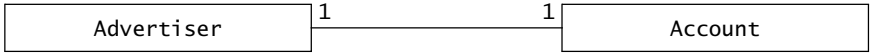
Note that many UML modeling tools accomplish the transformation of associations into references mechanically. However, even with a tool that accomplishes this transformation, it is nevertheless critical that you understand its rationale, as you have to deal with the generated code.

Unidirectional one-to-one associations. The simplest association is a unidirectional one-to-one association. For example (Figure 10-8), in ARENA, an Advertiser has a one-to-one association with an Account object that tracks all the charges accrued from displaying AdvertisementBanners. This association is unidirectional, as the Advertiser calls the operations of the Account object, but the Account never invokes operations of the Advertiser. In this case, we map this association to code using a reference from the Advertiser to the Account. That is, we add a field to Advertiser named account of type Account.

Creating the association between Advertiser and Account translates to setting the account field to refer to the correct Account object. Because each Advertiser object is associated with exactly one Account, a null value for the account attribute can only occur when a Advertiser object is being created. Otherwise, a null account is considered an error. Since the reference to the Account object does not change over time, we make the account field private and add a public Advertiser.getAccount() method. This prevents callers from accidentally modifying the account field.

Bidirectional one-to-one associations. The direction of an association often changes during the development of the system. Unidirectional associations are simple to realize. Bidirectional associations are more complex and introduce mutual dependencies among classes. Assume that we modify the Account class so that the display name of the Account is computed from the name of the Advertiser. In this case, an Account needs to access its corresponding Advertiser object. Consequently, the association between these two objects must be bidirectional

Object design model before transformation



Source code after transformation

```
public class Advertiser {
    private Account account;
    public Advertiser() {
        account = new Account();
    }
    public Account getAccount() {
        return account;
    }
}
```

Figure 10-8 Realization of a unidirectional, one-to-one association (UML class diagram and Java).

(Figure 10-9). We add an `owner` attribute to `Account` in the Java source code, but this is not sufficient: by adding a second attribute to realize the association, we introduce redundancy into the model. We need to ensure that if a given `Account` has a reference to a specific `Advertiser`, the `Advertiser` has a reference to that same `Account`. In this case, as the `Account` object is created by the `Advertiser` constructor, we add a parameter to the `Account` constructor to initialize the `owner` field to the correct value. Thus, the initial values for both fields are specified

Object design model before transformation

Source code after transformation

```
public class Advertiser {
    /* The account field is initialized
     * in the constructor and never
     * modified. */
    private Account account;

    public Advertiser() {
        account = new Account(this);
    }
    public Account getAccount() {
        return account;
    }
}
```

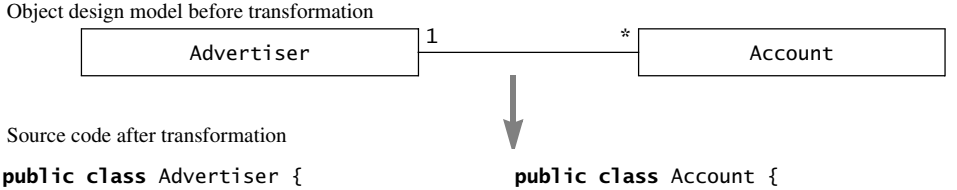
```
public class Account {
    /* The owner field is initialized
     * during the constructor and
     * never modified. */
    private Advertiser owner;

    public Account(Advertiser owner) {
        this.owner = owner;
    }
    public Advertiser getOwner() {
        return owner;
    }
}
```

Figure 10-9 Realization of a bidirectional one-to-one association (UML class diagram and Java excerpts).

in the same statement in the Advertiser constructor. Moreover, we make the owner field of Account private and add a public method to get its value. Since neither the Advertiser class nor the Account class modifies the field anywhere else, this ensures that both reference attributes remain consistent. Note that this assumption is not enforceable with the programming language constraints. The developer needs to document this assumption by writing a one-line comment immediately before the account and owner fields.

In Figure 10-9, both the Account and the Advertiser classes must be recompiled and tested whenever we change either class. With a unidirectional association from the Advertiser class to the Account class, the Account class would not be affected by changes to the Advertiser class. Bidirectional associations, however, are usually necessary in the case of classes that need to work together closely. The choice between unidirectional or bidirectional associations is a trade-off to be evaluated in each specific context. To make the trade-off easier, we can systematically make all attributes private and provide corresponding getAttribute() and setAttribute() operations to access the reference. This minimizes changes to APIs when changing a unidirectional association to bidirectional or vice versa.

One-to-many associations. One-to-many associations cannot be realized using a single reference or a pair of references. Instead, we realize the "many" part using a collection of references. For example, assume that an Advertiser can have several Accounts to track the expenses accrued by AdvertisementBanners for different products. In this case, the Advertiser object has a one-to-many association with the Account class (Figure 10-10). Because Accounts have no specific order and because an Account can be part of an Advertiser at most once, we



Source code after transformation

```
public class Advertiser {
    private Set accounts;
    public Advertiser() {
        accounts = new HashSet();
    }
    public void addAccount(Account a) {
        accounts.add(a);
        a.setOwner(this);
    }
    public void removeAccount
            (Account a) {
        accounts.remove(a);
        a.setOwner(null);
    }
}
```

```
public class Account {
    private Advertiser owner;
    public void setOwner
            (Advertiser newOwner) {
        if (owner != newOwner) {
            Advertiser old = owner;
            owner = newOwner;
            if (newOwner != null)
              newOwner.addAccount(this);
            if (old != null)
              old.removeAccount(this);
        }
    }
}
```

Figure 10-10 Realization of a bidirectional, one-to-many association (UML class diagram and Java).

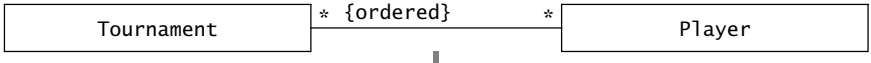
use a set of references, called accounts, to model the "many" part of the association. Moreover, we decide to realize this association as a bidirectional association, and so add the addAccount(), removeAccount(), and setOwner() methods to the Advertiser and Account classes to update the accounts and owner fields.

As in the one-to-one example, the association must be initialized when Advertiser and Account objects are created. However, since an Advertiser can have a varying number of Accounts, the Advertiser object does not invoke the Account constructor. Instead, a control object for creating and archiving Accounts is responsible for invoking the constructor.

Note that the collection on the "many" side of the association depends on the constraints on the association. For example, if the Accounts of an Advertiser must be ordered, we need to use a List instead of a Set. To minimize changes to the interface when association constraints change, we can set the return type of the getAccounts() method to Collection, a common superclass of List and Set.

Many-to-many associations. In this case, both end classes have fields that are collections of references and operations to keep these collections consistent. For example, the Tournament class of ARENA has an ordered many-to-many association with the Player class. This association is realized by using a List attribute in each class, which is modified by the operations addPlayer(), removePlayer(), addTournament(), and removeTournament() (Figure 10-11). We already identified acceptPlayer() and removePlayer() operations in the object design model (see Figure 9-11). We rename acceptPlayer() to addPlayer() to maintain consistency with the code generated for other associations.

Object design model before transformation



Source code after transformation

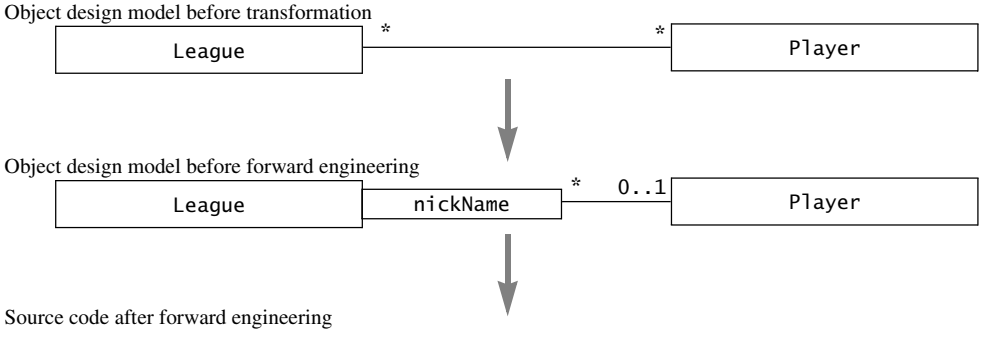
```
public class Tournament {
    private List players;
    public Tournament() {
        players = new ArrayList();
    }
    public void addPlayer(Player p) {
        if (!players.contains(p)) {
            players.add(p);
            p.addTournament(this);
        }
    }
}
```

```
public class Player {
    private List tournaments;
    public Player() {
        tournaments = new ArrayList();
    }
    public void addTournament
            (Tournament t) {
        if (!tournaments.contains(t)) {
            tournaments.add(t);
            t.addPlayer(this);
        }
    }
}
```

Figure 10-11 Realization of a bidirectional, many-to-many association (UML class diagram and Java).

As in the previous example, these operations ensure that both `Lists` are consistent. In the event the association between `Tournament` and `Player` should be unidirectional, we could then remove the `tournaments` attribute and its related methods, in which case a unidirectional many-to-many association or a unidirectional one-to-many association are very similar and difficult to distinguish at the object interface level.

Qualified associations. As we saw in Chapter 2, *Modeling with UML*, qualified associations are used to reduce the multiplicity of one "many" side in a one-to-many or a many-to-many association. The qualifier of the association is an attribute of the class on the "many" side of the association, such as a name that is unique within the context of the association, but not necessarily globally unique. For example, consider the association between `League` and `Player` (Figure 10-12). It is originally a many-to-many association (a `League` involves many `Players`, a `Player` can take part in many `Leagues`). To make it easier to identify `Players` within a `League`, `Players` can choose a short nickname that must be unique within the `League`. However, the `Player` can choose different nicknames in different `Leagues`, and the nicknames do not need to be unique globally within an `Arena`.



Source code after forward engineering

```
public class League {
    private Map players;

    public void addPlayer
            (String nickName, Player p) {
    if (!players.containsKey(nickName))
    {
        players.put(nickName, p);
        p.addLeague(nickName, this);
    }
    }
}
```

```
public class Player {
    private Map leagues;

    public void addLeague
            (String nickName, League l) {
        if (!leagues.containsKey(l)) {
            leagues.put(l, nickName);
            l.addPlayer(nickName, this);
        }
    }
}
```

Figure 10-12 Realization of a bidirectional qualified association (UML class diagram; arrow denotes the successive transformations).

Qualified associations are realized differently from the way one-to-many and many-to-many associations are realized. The main difference is that we use a `Map` object to represent the qualified end, as opposed to a `List` or a `Set`, and we pass the qualifier as a parameter in the operations to access the other end of the association. To continue our example, consider the association between `League` and `Player`. We realize this qualified association by creating a private `players` attribute in `League` and a `leagues` attribute in `Player`. The `players` attribute is a `Map` indexed by the nickname of the `Player` within the `League`. Because the nickname is stored in the `Map`, a specific `Player` can have different nicknames across `Leagues`. The `players` attribute is modified with the operations `addPlayer()` and `removePlayer()`. A specific `Player` is accessed with the `getPlayer()` with a specific `nickName`, which reduces the need for iterating through the `Map` to find a specific `Player`. The other end of the association is realized with a `Set`, as before.

Associations classes. In UML, we use an association class to hold the attributes and operations of an association. For example, we can represent the `Statistics` for a `Player` within a `Tournament` as an association class, which holds statistics counters for each `Player/Tournament` combination (Figure 10-13). To realize such an association, we first transform the association class into a separate object and a number of binary associations. Then we can use the techniques discussed earlier to convert each binary association to a set of reference attributes. In

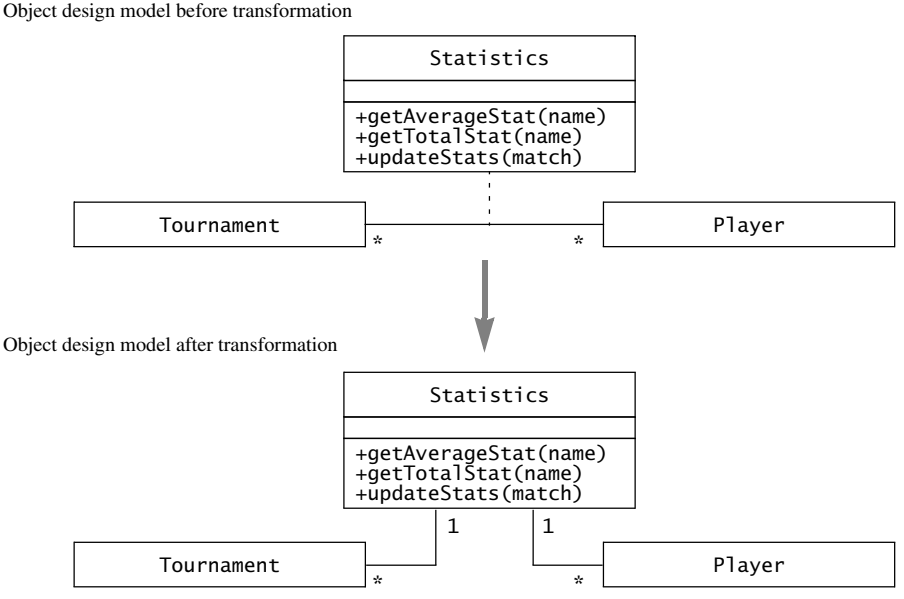


Figure 10-13 Transformation of an association class into an object and two binary associations (UML class diagram).

Section 10.6, we revisit this case and describe additional mappings for realizing association classes.

Once associations have been mapped to fields and methods, the public interface of classes is relatively complete and should change only as a result of new requirements, discovered bugs, or refactoring.

10.4.3 Mapping Contracts to Exceptions

Object-oriented languages that include constraints, such as Eiffel, can automatically check contracts and raise exceptions when a contract is violated. This enables a class user to detect bugs associated with incorrect assumptions about the used class. In particular, this is useful when developers reuse a set of classes to discover boundary cases. Raising exceptions when postconditions are violated enables class implementors to catch bugs early, to identify precisely the operation in which the violation occurred, and to correct the offending code.

Unfortunately, many object-oriented languages, including Java, do not provide built-in support for contracts. However, we can use their exception mechanisms as building blocks for signaling and handling contract violations. In Java, we raise an exception with the **`throw`** keyword followed by an exception object. The exception object provides a place holder for storing information about the exception, usually an error message and a backtrace representing the call stack of the `throw`. The effect of throwing an exception interrupts the control flow and unwinds the call stack until a matching **`catch`** statement is found. The `catch` statement is followed by a parameter, which is bound to the exception object, and an exception handling block. If the exception object is of the same type of the parameter (or a subclass thereof), the `catch` statement matches and the exception handling block is executed.

For example, in Figure 10-14, let us assume that the `acceptPlayer()` operation of `TournamentControl` is invoked with a player who is already part of the `Tournament`. In this case, `TournamentControl.addPlayer()` throws an exception of type `KnownPlayer`, which is caught by the caller, `TournamentForm.addPlayer()`, which forwards the exception to the `ErrorConsole` class, and then proceeds with the next `Player`. The `ErrorConsole` boundary object then displays a list of error messages to the user.

A simple mapping would be to treat each operation in the contract individually and to add code within the method body to check the preconditions, postconditions, and invariants relevant to the operation:

- *Checking preconditions*. Preconditions should be checked at the beginning of the method, before any processing is done. There should be a test that checks if the precondition is true and raises an exception otherwise. Each precondition corresponds to a different exception, so that the client class can not only detect that a violation occurred, but also identify which parameter is at fault.
- *Checking postconditions*. Postconditions should be checked at the end of the method, after all the work has been accomplished and the state changes are finalized. Each

postcondition corresponds to a Boolean expression in an if statement that raises an exception if the contract is violated. If more than one postcondition is not satisfied, only the first detection is reported.

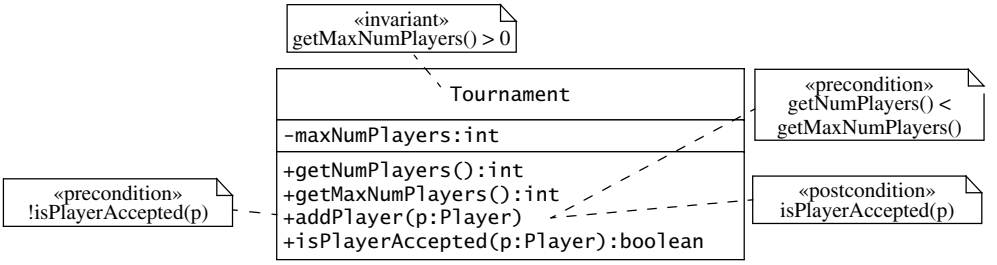
- *Checking invariants*. When treating each operation contract individually, invariants are checked at the same time as postconditions.
- *Dealing with inheritance*. The checking code for preconditions and postconditions should be encapsulated into separate methods that can be called from subclasses.

```
public class TournamentControl {
    private Tournament tournament;
    public void addPlayer(Player p) throws KnownPlayerException {
        if (tournament.isPlayerAccepted(p)) {
            throw new KnownPlayerException(p);
        }
        //... Normal addPlayer behavior
    }
}
public class TournamentForm {
    private TournamentControl control;
    private List players;
    public void processPlayerApplications() {
        // Go through all the players who applied for this tournament
        for (Iterator i = players.iterator(); i.hasNext();) {
            try {
                // Delegate to the control object.
                control.acceptPlayer((Player)i.next());
            } catch (KnownPlayerException e) {
                // If an exception was caught, log it to the console, and
                // proceed to the next player.
                ErrorConsole.log(e.getMessage());
            }
        }
    }
}
```

Figure 10-14 Example of exception handling in Java. `TournamentForm` catches exceptions raised by `Tournament` and `TournamentControl` and logs them into an error console for display to the user.

A systematic application of the above rules to the `Tournament.addPlayer()` contract yields the code in Figure 10-15.

If we mapped every contract following the above steps, we would ensure that all preconditions, postconditions, and invariants are checked for every method invocation, and that violations are detected within one method invocation. While this approach results in a robust system (assuming the checking code is correct), it is not realistic:




```
public class Tournament {
//...
    private List players;

    public void addPlayer(Player p)
        throws KnownPlayer, TooManyPlayers, UnknownPlayer,
            IllegalNumPlayers, IllegalMaxNumPlayers
    {
        // check precondition!isPlayerAccepted(p)
        if (isPlayerAccepted(p)) {
            throw new KnownPlayer(p);
        }
        // check precondition getNumPlayers() < maxNumPlayers
        if (getNumPlayers() == getMaxNumPlayers()) {
            throw new TooManyPlayers(getNumPlayers());
        }
        // save values for postconditions
        int pre_getNumPlayers = getNumPlayers();

        // accomplish the real work
        players.add(p);
        p.addTournament(this);

        // check post condition isPlayerAccepted(p)
        if (!isPlayerAccepted(p)) {
            throw new UnknownPlayer(p);
        }
        // check post condition getNumPlayers() = @pre.getNumPlayers() + 1
        if (getNumPlayers() != pre_getNumPlayers + 1) {
            throw new IllegalNumPlayers(getNumPlayers());
        }
        // check invariant maxNumPlayers > 0
        if (getMaxNumPlayers() <= 0) {
            throw new IllegalMaxNumPlayers(getMaxNumPlayers());
        }
    }
    //...
}
```

Figure 10-15 A complete implementation of the `Tournament.addPlayer()` contract.

- *Coding effort*. In many cases, the code required for checking preconditions and postconditions is longer and more complex than the code accomplishing the real work. This results in increased effort that could be better spent in testing or code clean-up.
- *Increased opportunities for defects*. Checking code can also include errors, increasing testing effort. Worse, if the same developer writes the method and the checking code, it is highly probable that bugs in the checking code mask bugs in the actual method, thereby reducing the value of the checking code.
- *Obfuscated code*. Checking code is usually more complex than its corresponding constraint and difficult to modify when constraints change. This leads to the insertion of many more bugs during changes, defeating the original purpose of the contract.
- *Performances drawback*. Checking systematically all contracts can significantly slow down the code, sometimes by an order of magnitude. Although correctness is always a design goal, response time and throughput design goals would not be met.

Hence, unless we have a tool for generating checking code automatically, such as iContract [Kramer, 1998], we need to adopt a pragmatic approach and evaluate the above trade-offs in the project context. Remember that contracts support communication among developers, consequently, exception handling of contract violations should focus on interfaces between developers. Below are heuristics to evaluate these trade-offs:

Heuristics for mapping contracts to exceptions

- *Omit checking code for postconditions and invariants*. Checking code is usually redundant with the code accomplishing the functionality of the class, and is written by the developer of the method. It is not likely to detect many bugs unless it is written by a separate tester.
- *Focus on subsystem interfaces* and omit the checking code associated with private and protected methods. System boundaries do not change as often as internal interfaces and represent a boundary between different developers.
- *Focus on contracts for components with the longest life*, that is, on code most likely to be reused and to survive successive releases. Entity objects usually fulfill these criteria, whereas boundary objects associated with the user interface do not.
- *Reuse constraint checking code*. Many operations have similar preconditions. Encapsulate constraint checking code into methods so that they can be easily invoked and so that they share the same exception classes.

In all cases, the checking code should be documented with comments describing the constraints checked, both in English and in OCL. In addition to making the code more readable, this makes it easier to modify the checking code correctly when a constraint changes.

10.4.4 Mapping Object Models to a Persistent Storage Schema

So far, we have treated persistent objects like all other objects. However, object-oriented programming languages do not usually provide an efficient way to store persistent objects. In this case, we need to map persistent objects to a data structure that can be stored by the persistent data management system decided during system design, in most cases, either a database or a set of files. For object-oriented databases, no transformations need be done, since there is a one-to-one mapping between classes in the object model and classes in the object-oriented database. However, for relational databases and flat files, we need to map the object model to a storage schema and provide an infrastructure for converting from and to persistent storage. In this section, we look at the steps involved in mapping an object model to a relational database using Java and database schemas.

A **schema** is a description of the data, that is, a meta-model for data [Date, 2004]. In UML, class diagrams are used to describe the set of valid instances that can be created by the source code. Similarly, in relational databases, the database schema describes the valid set of data records that can be stored in the database. Relational databases store both the schema and the data. Relational databases store persistent data in the form of tables (also called *relations* in the database literature). A table is structured in columns, each of which represents an **attribute**. For example, in Figure 10-16, the `User` table has three columns, `firstName`, `login`, and `email`. The rows of the table represent data records, with each cell in the table representing the value of the attribute for the data record in that row. In Figure 10-16, the `User` table contains three data records each representing the attributes of specific users Alice, John, and Bob.

A **primary key** of a table is a set of attributes whose values uniquely identify the data records in a table. The primary key is used to refer unambiguously to a specific data record when inserting, updating, or removing it. For example, in Figure 10-16, the `login` attribute represents a unique user name within an `Arena`. Hence, the `login` attribute can be used as a primary key. Note, however, the `email` attribute is also unique across all users in the table. Hence, the `email` attribute could also be used as a primary key. Sets of attributes that could be used as a primary

User table

Primary key

firstName	login	email
“alice”	“am384”	“am384@mail.org”
“john”	“js289”	“john@mail.de”
“bob”	“bd”	“bobd@mail.ch”

Candidate key Candidate key

Figure 10-16 An example of a relational table, with three attributes and three data records.

League table

name	login
“tictactoeNovice”	“am384”
“tictactoeExpert”	“am384”
“chessNovice”	“js289”

Foreign key referencing User table

Figure 10-17 An example of a foreign key. The `owner` attribute in the `League` table refers to the primary key of the `User` table in Figure 10-16.

key are called **candidate keys**. Only the actual candidate key that is used in the application to identify data records is the primary key.

A **foreign key** is an attribute (or a set of attributes) that references the primary key of another table. A foreign key links a data record in one table with one or more data records in another table. In Figure 10-17, the table `League` includes the foreign key `owner` that references the `login` attribute in the `User` table in Figure 10-16. Alice is the owner of the `tictactoeNovice` and `tictactoeExpert` leagues and John is the owner of the `chessNovice` league.

Mapping classes and attributes

When mapping the persistent objects to relational schemata, we focus first on the classes and their attributes. We map each class to a table with the same name. For each attribute, we add a column in the table with the name of the attribute in the class. Each data record in the table corresponds to an instance of the class. By keeping the names in the object model and the relational schema consistent, we provide traceability between both representations and make future changes easier.

When mapping attributes, we need to select a data type for the database column. For primitive types, the correspondence between the programming language type and the database type is usually trivial (e.g., the Java `Date` type maps to the `datetime` type in SQL). However, for other types, such as `String`, the mapping is more complex. The type text in SQL requires a specified maximum size. For example, when mapping the `ARENA` `User` class, we could arbitrarily limit the length of first names to 25 characters, enabling us to use a column of type `text[25]`. Note that we have to ensure that users’ first names comply with this new constraint by adding preconditions and checking code in the entity and boundary objects.

Next, we focus on the primary key. There are two options when selecting a primary key for the table. The first option is to identify a set of class attributes that uniquely identifies the object. The second option is to add a unique identifier attribute that we generate.

For example, in Figure 10-16, we use the login name of the user as a primary key. Although this approach is intuitive, it has several drawbacks. If the value of the login attribute changes, we need to update all tables in which the user login name occurs as a foreign key. Also, selecting attributes from the application domain can make it difficult to change the database schema when the application domain changes. For example, in the future, we could use a single table to store users from different `Arenas`. As login names are unique only within a single `Arena`, we would need to add the name of the `Arena` in the primary key.

The second option is to use an arbitrarily unique identifier (`id`) attribute as a primary key. We generate the `id` attribute for each object and can guarantee that it is unique and will not change. Some database management systems provide features for automatically generating `ids`. This results in a more robust schema and primary and foreign keys that consist of one column.

For example, let us focus on the `User` class in `ARENA` (Figure 10-18). We map it to a `User` table with four columns: `id`, `firstName`, `login`, and `email`. The type of the `id` column is a `long` integer that we increment every time we create a new object.

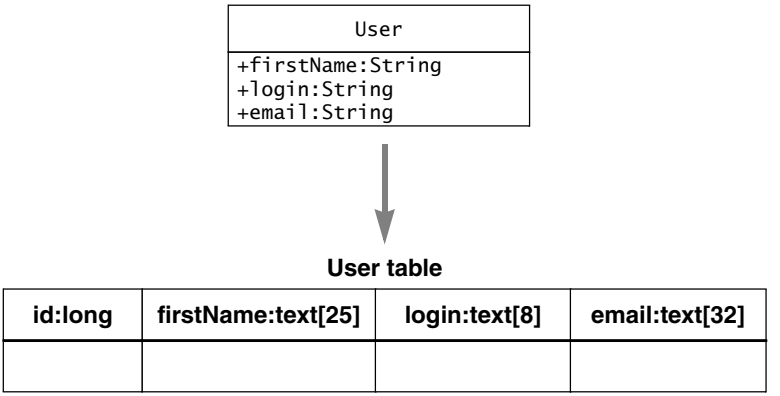


Figure 10-18 Forward engineering of the `User` class to a database table.

Mapping associations

After having mapped the classes to relational tables, we now turn to the mapping of associations. The mapping of associations to a database schema depends on the multiplicity of the association. One-to-one and one-to-many associations are implemented as a so-called **buried association** [Blaha & Premerlani, 1998], using a foreign key. Many-to-many associations are implemented as a separate table.

Buried associations. Associations with multiplicity one can be implemented using a foreign key. For one-to-many associations, we add a foreign key to the table representing the class on the "many" end. For all other associations, we can select either class at the end of the association. For example (Figure 10-19), consider the one-to-many association between `LeagueOwner` and

League. We map this association by adding a owner column to the League table referring to the primary key of the LeagueOwner table. The value of the owner column is the value of the id (i.e., the primary key) of the corresponding league. If there are multiple Leagues owned by the same LeagueOwner, multiple data records of the League table have the id of the owner as value for this column. For associations with a multiplicity of zero or one, a null value indicates that there are no associations for the data record of interest.

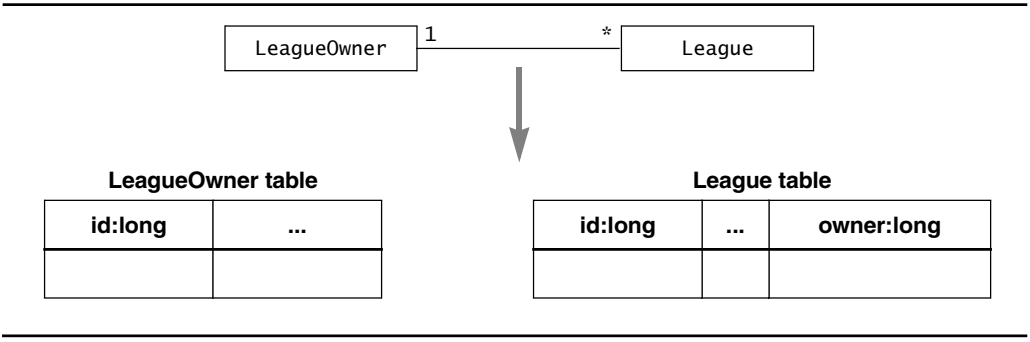


Figure 10-19 Mapping of the LeagueOwner/League association as a buried association.

Separate table. Many-to-many associations are implemented using a separate two-column table with foreign keys for both classes of the association. We call this the *association table*. Each row in the association table corresponds to a link between two instances. For example, we map the many-to-many Tournament/Player association to an association table with two columns: one for the id of the Tournaments, the other for the id of the Players. If a player is part of multiple tournaments, each player/tournament association will have a separate data record. Similarly, if a tournament includes multiple players, each player will have a separate data record. The association table in Figure 10-20 contains two links representing the membership of "alice" and "john" in the "novice" Tournament.

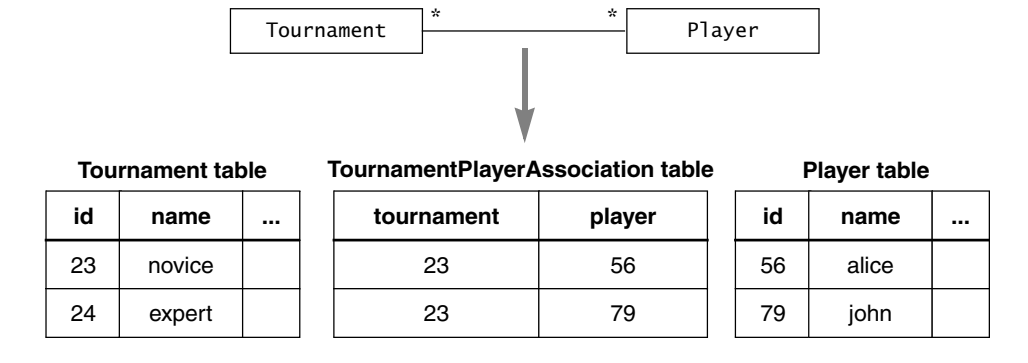


Figure 10-20 Mapping of the Tournament/Player association as a separate table.

Note that a one-to-one and one-to-many association could be realized with an association table instead of a buried association. Using a separate table to realize all associations results in a database schema that is modifiable. For example, if we change the multiplicity of a one-to-many association to a many-to-many association, we do not need to change the database schema. Of course, this increases the overall number of tables in the schema and the time to traverse the association. In general, we need to evaluate this trade-off in the context of the application, examining whether the multiplicity of the association is likely to change or if response time is a critical design goal.

Mapping inheritance relationships

Relational databases do not directly support inheritance, but there are two main options for mapping an inheritance relationship to a database schema. In the first option, called *vertical mapping*, similar to a one-to-one association, each class is represented by a table and uses a foreign key to link the subclass tables to the superclass table. In the second option, called *horizontal mapping*, the attributes of the superclass are pushed down into the subclasses, essentially duplicating columns in the tables corresponding to subclasses.

Vertical mapping. Given an inheritance relationship, we map the superclass and subclasses to individual tables. The superclass table includes a column for each attribute defined in the superclass. The superclass includes an additional column denoting the subclass that corresponds to the data record. The subclass tables include a column for each attribute defined in the superclass. All tables share the same primary key, that is, the identifier of the object. Data records in the superclass and subclass tables with the same primary key value refer to the same object.

For example, in Figure 10-21, the classes `User`, `LeagueOwner`, and `Player` are each mapped to a table with columns for the attributes of the classes. The `User` table includes an additional column, `role`, which denotes the class of the data record. The three tables share the same primary key, that is, if data records have the same `id` value in each table, they correspond to the same object. The `User` Zoe, for example, is a `LeagueOwner`, who can lead at most 12 `Leagues`. Zoe's name is stored in the `User` table; the maximum number of leagues is stored in the `LeagueOwner` table. Similarly, the `User` John is a `Player` who has 126 credits (i.e., the number of `Matches` that John can play before renewing his membership). John's name is stored in the `User` table; his number of credits is stored in the `Player` table.

In general, to retrieve an object from the database, we need to examine first the data record in the superclass table. This record includes the attribute values for the superclass (e.g., "zoe", the name of the user) and the subclass of the instance. We use the same object `id` (e.g., "56") to query the subclass indicated by the role attribute (i.e., "LeagueOwner"), and retrieve the remainder of the attribute values (e.g., "maxNumLeagues = 12"). In case of several levels of inheritance, we repeat the same procedure and reconstruct the object with individual queries to each table.

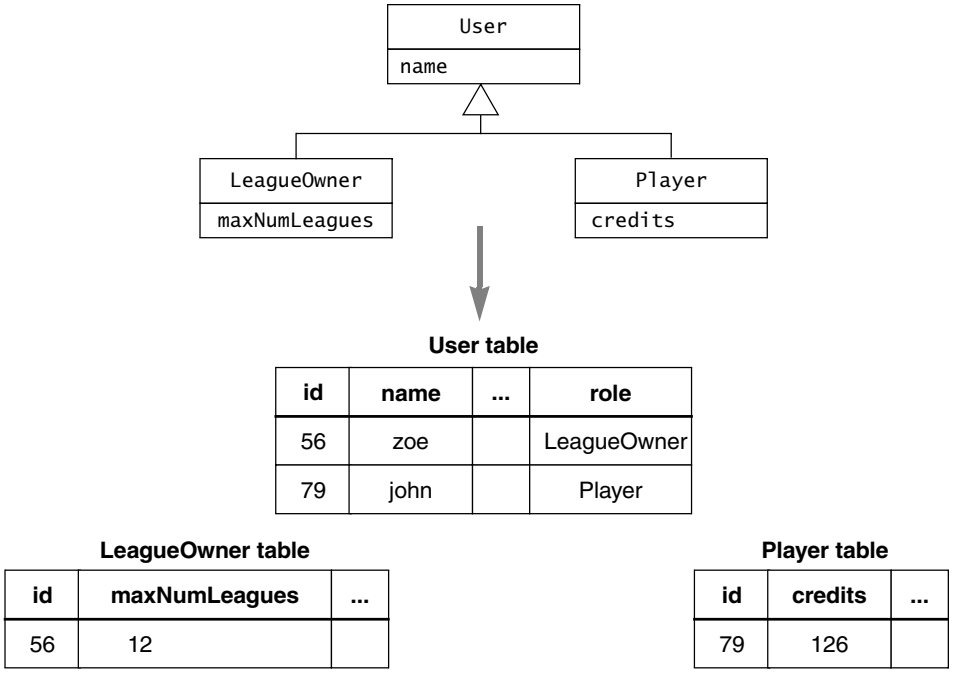


Figure 10-21 Realizing the `User` inheritance hierarchy with a separate table.

Horizontal mapping. Another way to realize inheritance is to push the attributes of the superclass down into the subclasses, effectively removing the need for a superclass table. In this case, each subclass table duplicates the columns of the superclass.

For example, in the case of the `User` inheritance hierarchy, we create a table for `LeagueOwners` and a table for `Players` (Figure 10-22). Each table includes a column for its own attributes and for the attributes of the `User` class. In this case, we need a single query to retrieve all attribute values for a single object.

The trade-off between using a separate table for superclasses and duplicating columns in the subclass tables is between modifiability and response time. If we use a separate table, we can add attributes to the superclass simply by adding a column to the superclass table. When adding a subclass, we add a table for the subclass with a column for each attribute in the subclass. If we duplicate columns, modifying the database schema is more complex and error prone. The advantage of duplicating columns is that individual objects are not fragmented across a number of tables, which results in faster queries. For deep inheritance hierarchies, this can represent a significant performance difference.

In general, we need to examine the likelihood of changes against the performance requirements in the specific context of the application.

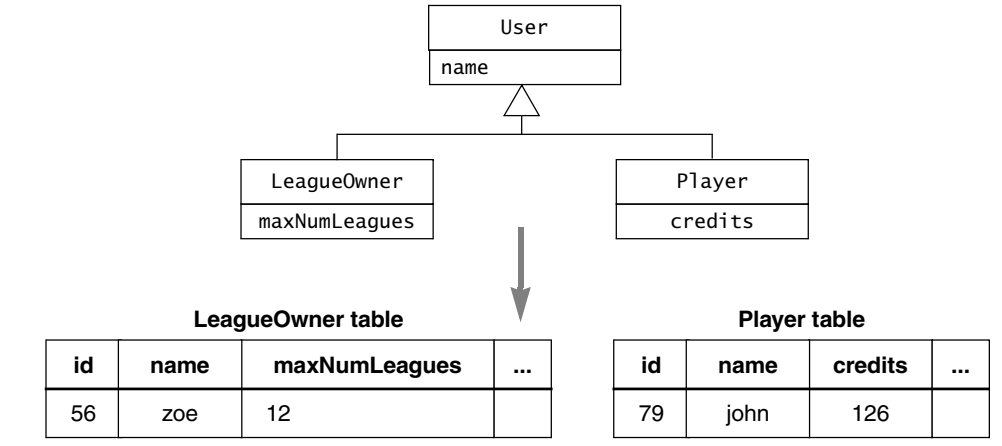


Figure 10-22 Realizing the `User` inheritance hierarchy by duplicating columns.

10.5 Managing Implementation

10.5.1 Documenting Transformations

Transformations enable us to improve specific aspects of the object design model and to convert it into source code. By providing systematic recipes for recurring situations, transformations enable us to reduce the amount of effort and the overall number of errors in the source code. However, to retain this benefit throughout the lifetime of the system, we need to document the application of transformations so that they can be consistently reapplied in the event of changes to the object design model or the source code.

Reverse engineering attempts to alleviate this problem by allowing us to reconstruct the object design model from the source code. If we could maintain a one-to-one mapping between the source code and the object design mode, we would not need any documentation: the tools at hand would automatically apply selected transformations and mirror changes in the source code and the object design model. However, most useful transformations, including those described in this chapter, are not one-to-one mappings. As a result, information is lost in the process of applying the transformation. For example:

- *Association multiplicity and collections*. Unidirectional one-to-many associations and many-to-many associations map to the same source code. A CASE tool that reverse-engineers the corresponding source code usually selects the least restrictive case (i.e., a many-to-many association). In general, information about association multiplicity is distributed in several places in the source code, including checking code in the boundary objects.

- *Association multiplicity and buried associations*. One-to-many associations and one-to-one associations implemented as a buried association in a database schema suffer from the same problem. Worse, when all associations are realized as separate tables, all information about association multiplicity is lost.
- *Postconditions and invariants*. When mapping contracts to exception-handling code (Section 10.4.3), we generate checking code only for preconditions. Postconditions and invariants are not mapped to source code. The object specification and the system become quickly inconsistent when postconditions or invariants are changed, but not documented.

These challenges boil down to finding conventions and mechanisms to keep the object design model, the source code, and the documentation consistent with each other. There is no single answer, but the following principles reduce consistency problems when applied systematically:

- *For a given transformation, use the same tool*. If you are using a modeling tool to map associations to code, use the same tool when you change association multiplicities. Modern modeling tools generate markers as source code comments to enable the repetitive generation of code from the same model. However, this mapping can easily break when developers use interchangeably a text editor or the modeling tool to change associations. Similarly, if you generate constraint-checking code with a tool, regenerate the checking code when the constraint is changed.
- *Keep the contracts in the source code, not in the object design model*. Contracts describe the behavior of methods and restrictions on parameters and attributes. Developers change the behavior of an object by modifying the body of a method, not by modifying the object design model. By keeping the constraint specifications as source code comments, they are more likely to be updated when the code changes.
- *Use the same names for the same objects*. When mapping an association to source code or a class to a database schema, use the same names on both sides of the transformation. If the name is changed in the model, change it in the source code. By using the same names, you provide traceability among the models and make it easier for developers to identify both ends of the transformation. This also emphasizes the importance of identifying the right names for classes during analysis, before any transformations are applied, to minimize the effort associated with renaming.
- *Make transformations explicit*. When transformations are applied by hand, it is critical that the transformation is made explicit in some form so that all developers can apply the transformation the same way. For example, transformations for mapping associations to collections should be documented in a coding conventions guide so that, when two developers apply the same transformation, they produce the same code. This also makes it easier for developers to identify transformations in the source code. As

usual, the commitment of developers to use standard conventions is more important than the actual conventions.

10.5.2 Assigning Responsibilities

Several roles collaborate to select, apply, and document transformations and the conversion of the object design model into source code:

- The **core architect** selects the transformations to be systematically applied. For example, if it is critical that the database schema is modifiable, the core architect decides that all associations should be implemented as separate tables.
- The **architecture liaison** is responsible for documenting the contracts associated with subsystem interfaces. When such contracts change, the architecture liaison is responsible for notifying all class users.
- The **developer** is responsible for following the conventions set by the core architect and actually applying the transformations and converting the object design model into source code. Developers are responsible for maintaining up-to-date the source code comments with the rest of the models.

Identifying and applying transformations the first time is relatively trivial. The key challenge is in reapplying transformations after a change occurs. Hence, when assigning responsibilities, each role should understand who should be notified in the event of changes.

10.6 ARENA Case Study

We now apply the concepts and methods described in this chapter to a more extensive example from the ARENA system. We focus on the classes surrounding `Statistics`, which is responsible for tracking general and game-specific statistics for each *Game*, `Player`, `Tournament`, and `League`. The challenge in designing and realizing the associations among these classes is to find a reusable solution for many types of statistics, present and future, so that as little code as possible must be written when new *Games* or *TournamentStyles* are introduced.

First, we describe in detail the `Statistics` class. Next, we map the associations between the `Statistics` object and the other objects to collections. Next, we map the contracts of the `Statistics` class to exceptions. Finally, we design a database schema for these classes and associations.

10.6.1 ARENA Statistics

A `Statistics` instance is responsible for tracking a number of running counters within the context of a *Game* and for a number of scopes. For example, a `Spectator` should be able to view the `Statistics` associated with a specific `Player` within the scope of a single `Tournament` (e.g., the average number of moves per `Match` by player John in the winter Tic Tac Toe tournament) or within the more general scope of a `League` (e.g., the average number of moves per match by

player John in the Tic Tac Toe novice league). Similarly, a Spectator should also be able to view average statistics over all Players in a League. The scope of a statistic are, from most general to most specific:

- All Matches in a *Game*
- All Matches in a League
- All Matches in a League played by a specific Player
- All Matches in a Tournament
- All Matches in a Tournament played by a specific Player.

Earlier (see Section 8.6.1), we addressed the issue of *Game* independence by defining Statistics as a product in an Abstract Factory pattern. For each new *Game* added to ARENA, developers are asked to refine the Statistics interface. For *Games* that do not need specialized Statistics, we provide a default implementation of the Statistics interface called DefaultStatistics. ARENA does not access the concrete *Game* and Statistics classes, thereby ensuring *Game* independence (Figure 10-23).

Our design goal is to keep the Statistics interface simple so that it can be easily specialized for different *Games*. In other words, the concrete Statistics class for a specific *Game* should only have to define the formulae for computing the *Game*-specific Statistics. Scopes

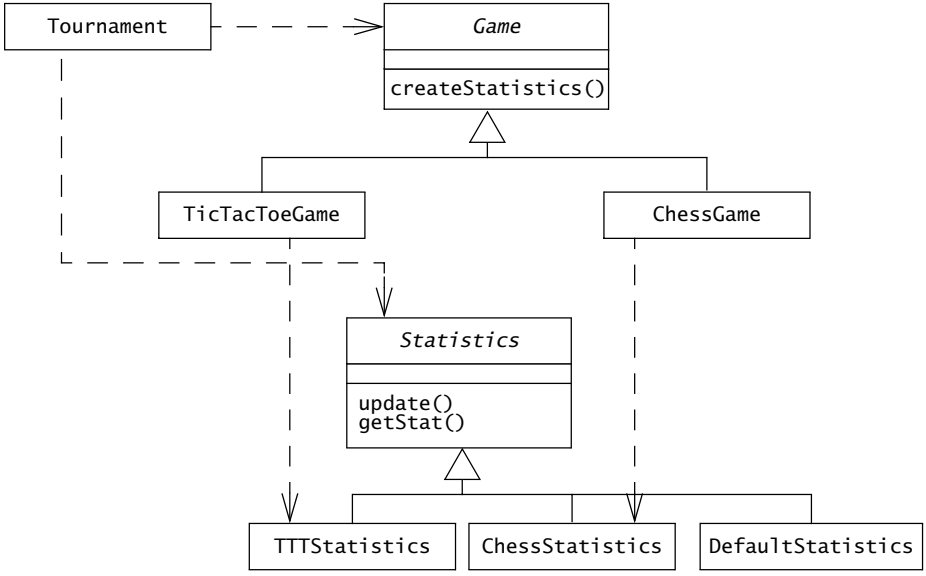


Figure 10-23 Statistics as a product in the *Game* Abstract Factory (UML class diagram).

should be handled by ARENA. Moreover, response time in ARENA has a higher priority than memory consumption.

This leads us, during object design, to the decision of computing all Statistics incrementally as each Match completes. We organize the individual counters we compute into Statistics objects, each representing a scope. Hence, there is a Statistics object for each *Game*, League, Tournament, and for each combination of Player/*Game*, Player/League, and Player/Tournament.

For example, let us assume that Player John takes part in the two Tournaments, t1 and t2, in the Tic Tac Toe novice League. He then moves on to the expert Tic Tac Toe League and takes part in one more Tournament, t3. Let us further assume that the Tic Tac Toe Statistics object tracks his win ratio, the ratio of the number of the *Games* he has won over all the *Games* he has played. Under these circumstances, six Statistics objects track John's win ratio:

- Three Statistics objects track John's win ratio for the Tournaments t1, t2, and t3.
- Two Statistics objects tracks John's win ratio over all his Tournaments within the novice and the expert Leagues, respectively.
- One for the Tic Tac Toe *Game*, tracking John's lifetime win ratio.

In UML, we can represent compactly this complex set of interactions with an N-ary association Statistics class relating the Player, the *Game*, the League, and the Tournament classes (Figure 10-24). For any given Statistics association, only one of *Game*, League, or Tournament is involved in the association, denoting whether the Statistics is value for all Leagues in a specific *Game*, for a single League, or for a single Tournament, respectively.

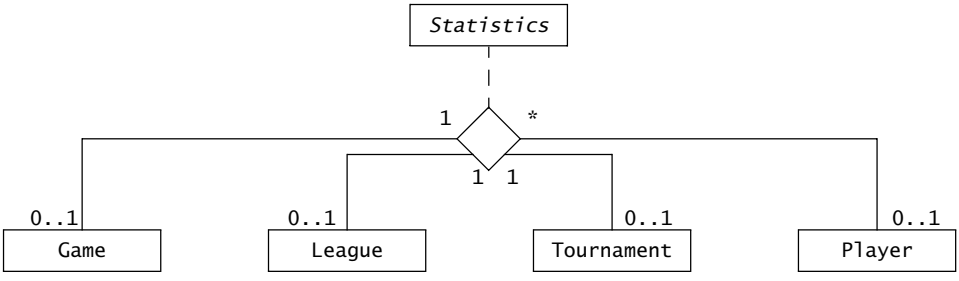


Figure 10-24 N-ary association class Statistics relating League, Tournament, and Player (UML class diagram).

10.6.2 Mapping Associations to Collections

We map the N-ary Statistics association to a Statistics Java class, whose sole purpose is to hold the values of the counters, and a singleton StatisticsVault, responsible for tracking the

links between the Statistics objects and the relevant *Game*, League, Tournament, and Player instances (Figure 10-25). We add operations to StatisticsVault for retrieving a specific Statistics given a scope. If the Statistics object of interest does not exist, the StatisticsVault creates it using the corresponding *Game.createStatistics()* method. Internally, the StatisticsVault uses a private HashMap to store the relationship between the combination of Player, *Game*, League, and Tournament and the corresponding Statistics object.

SimpleStatisticsVault depicted in Figure 10-25 is a direct mapping of the N-ary association of Figure 10-24. SimpleStatisticsVault does not accomplish any other task beyond maintaining the state of the association. TournamentControl invokes methods of SimpleStatisticsVault to retrieve the needed Statistics objects, and then invokes the Statistics.update() method as Matches complete. StatisticsView (a boundary object displaying statistics to the user) retrieves the needed Statistics based on the user selection and invokes the Statistics.getStat() method to retrieve the individual values of the counters. In both cases, two steps are needed, one to retrieve the correct Statistics object, the other to invoke the method that does the actual work.

Ideally, we would like to avoid this situation and design an interface that enables TournamentControl and StatisticsView to invoke only one method. This would simplify the method calls related to Statistics in both objects. It would also centralize the scope look-up logic into a single object, making future changes easier. To accomplish this, we refine the SimpleStatisticsVault into a Facade design pattern (see Figure 6-30). We merge the getStatisticsObject() methods with the methods offered by the Statistics object; that is, we provide an update() method for TournamentControl and a set of getStat() methods for

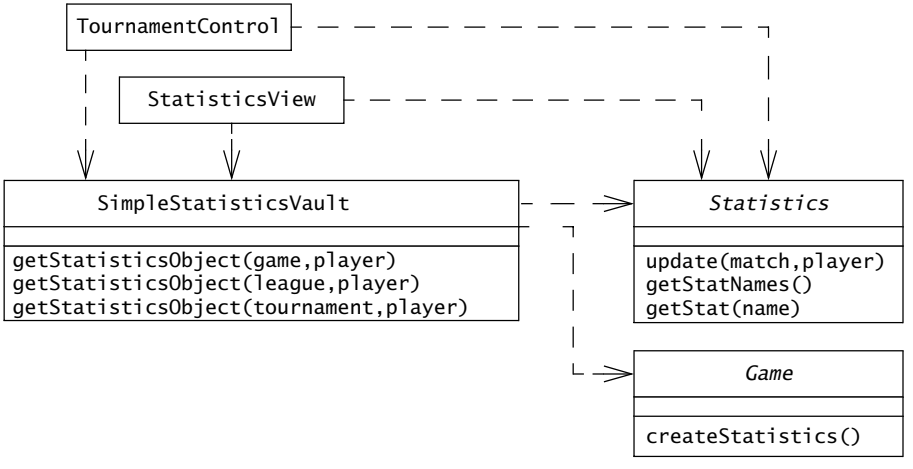


Figure 10-25 SimpleStatisticsVault object realizing the N-ary association of Figure 10-24.

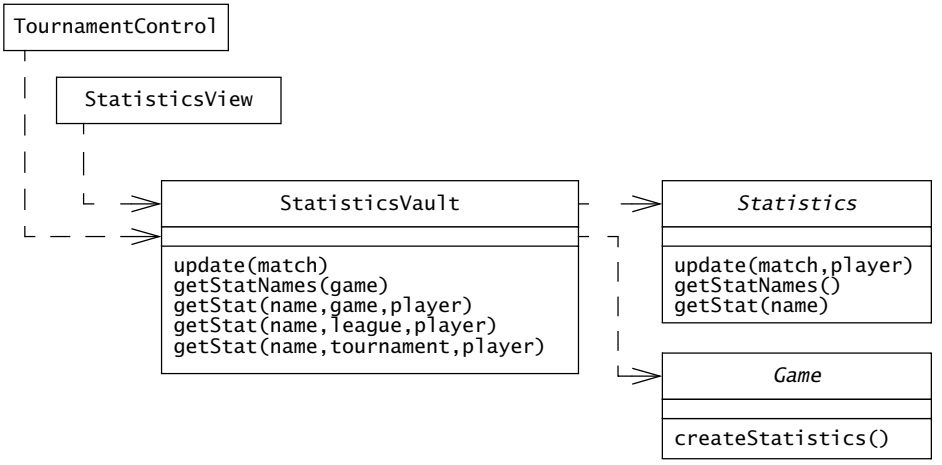


Figure 10-26 `StatisticsVault` as a Facade shielding the control and boundary objects from the Statistics storage and computation (UML class diagram).

`StatisticsView`. This results in a slightly more complex interface (Figure 10-26). However, the interface is still within 7 ± 2 methods, and all handling of `Statistics` objects is now centralized.

If we decide to add different scopes (e.g., match statistics), we will only need to add a method to `StatisticsVault`. Moreover, `TournamentControl` and `StatisticsView` are now completely decoupled from the individual `Statistics` objects, giving us the option of changing the way `Statistics` are stored, for example, by computing them on the fly or by caching them with a Proxy design pattern (Figure 10-7).

The role of the `Statistics` object is reduced to tracking counters (basically, name/value pairs). None of the classes `Statistics`, `Player`, `Tournament`, or `League` store data about the N-ary association. Consequently, when writing a specialized `Statistics` object for a new *`Game`*, the class extender need only focus on the formula for computing a statistic given a `Match` and a `Player`.

10.6.3 Mapping Contracts to Exceptions

Because we introduced a new interface, the `StatisticsVault` facade, in the `ARENA` object design model, we need to write its contract and relate it to the contracts of the existing classes. First, we focus on the constraints associated with the Statistics class and propagate them to the `StatisticsVault` class. The `Statistcs.getStat()` method assumes that the name passed in parameter is that of a statistic known by this type of object. Hence, we add one such constraint for each `getStat()` method in the `StatisticsVault`:

```
context StatisticsVault::getStat(name,game,player) pre:
    getStatNames()->includes(name)
context StatisticsVault::getStat(name,league,player) pre:
    getStatNames()->includes(name)
context StatisticsVault::getStat(name,tournament,player) pre:
    getStatNames()->includes(name)
```

Similarly, we add constraints to the `StatisticsVault.update()` method to reflect the constraint on the `Statistics.update()` method. In this case, we only need to ensure that the `match` is not `null` and has been completed.

```
context StatisticsVault::update(match) pre:
    match <> null and match.isCompleted()
```

Next, we need to examine the remaining parameters of the `StatisticsVault` methods and document any additional preconditions. We stipulate that the `player` parameter can be `null` (denoting the `Statistics` object for all players), but parameters specifying a `game`, `league`, and `tournament` cannot be `null`, because a `null` value, in this case, does not correspond to an application domain concept (e.g., no statistics are valid across all *Games*). In addition, if a player is specified, she must be related to the scope object (e.g., a player does not have statistics for a tournament in which she did not play).

```
context StatisticsVault::getStat(name,game,player) pre:
    game <> null and
        player <> null implies player.leagues.game->includes(game)
context StatisticsVault::getStat(name,league,player) pre:
    league <> null and
        player <> null implies league.players->includes(player)
context StatisticsVault::getStat(name,tournament,player) pre:
    tournament <> null and
        player <> null implies tournament.players->includes(player)
```

Finally, we map the above constraints to exceptions and checking code. For constraints that we propagated from the `Statistics` object, we simply forward the exceptions we receive from `Statistics` and omit the checking code. The `UnknownStatistic` is raised by `Statistics.getStat()` and forwarded by `StatisticsVault.getStat()` methods. The `InvalidMatch` and `MatchNotCompleted` exceptions are raised by `Statistics.update()` and forwarded by `StatisticsVault.update()` method. For the other constraints, we define a new exception, `InvalidScope`, for the cases where the `game`, `league`, or `tournament` parameters are `null` or are not related to the specified `player`. Finally, we add checking code at the beginning of the `StatisticsVault.getStat()` methods to check for `null` references and raise the `InvalidScope` exception, if needed. Figure 10-27 depicts the public interface of the `StatisticsVault` class.

```
public class StatisticsVault {
    public void update(Match m)
        throws InvalidMatch, MatchNotCompleted {...}

    public List getStatNames() {...}

    public double getStat(String name, Game g, Player p)
        throws UnknownStatistic, InvalidScope {...}

    public double getStat(String name, League l, Player p)
        throws UnknownStatistic, InvalidScope {...}

    public double getStat(String name, Tournament t, Player p)
        throws UnknownStatistic, InvalidScope {...}
}
```

Figure 10-27 Public interface of the `StatisticsVault` class (Java).

Note that, with the `InvalidScope` exception, we decided to cover three preconditions with one exception. This results in similar method declarations for all three `getStat()` methods, thereby making it easier for the calling class to handle violations of all three preconditions with the same handling code. In general, a set of overloaded methods should have similar interfaces since they implement the same operation for the different types of parameters.

10.6.4 Mapping the Object Model to a Database Schema

In the past two subsections, we transformed the N-ary `Statistics` association class (Figure 10-24) into a set of Java classes and operations. In this section, we start with the same object model and map it into a set of tables. As `Statistics` is an N-ary association class, we cannot use buried keys. We first map the association to a separate table, containing a foreign key for each of the association ends (*Game*, `League`, `Tournament`, `Player`). We then note that any given `Statistic` includes only one link to either a *Game*, a `League`, or a `Tournament`. Consequently, we can collapse the columns for *Game*, `League`, and `Tournament` into a single column denoting the `id` of the scope object and a column denoting the type of object (`scopetype`, Figure 10-28). We decide to encode the type of scope as a long value to save on storage space and on retrieval speed. The mapping between the long values and the actual class is done in the storage subsystem. This decision is a local optimization and does not affect the interface of the `StatisticsVault`. The `id` of the `Statistics` object is the primary key of the table.

As each `Statistic` has a variable number of name/value pairs, we use a separate table for the counter values, including one column for the `name` of the counter, one for the `value`, and a foreign key into the `Statistics` table (i.e., the `Statistics id`). The primary key for the `StatisticsCounter` table is the `Statistics id` and the counter `name`.

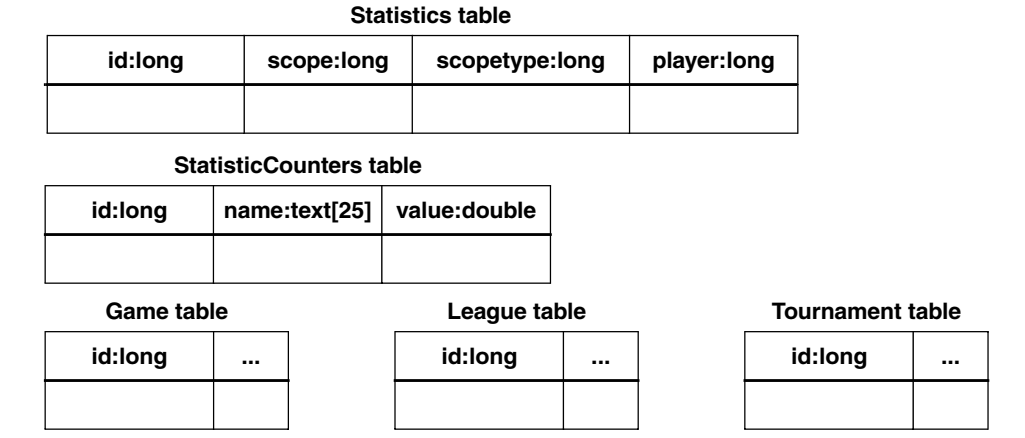


Figure 10-28 Database schema for the `Statistics` N-ary association of Figure 10-24.

Note that when designing the database schema for the `Statistics` association, we started with the object design model, not the Java classes that we generated in the previous section. There are several reasons for starting with the object design model:

- The object design model represents a view of the application domain and is less likely to change than the source code.
- When starting from the object design model, the database schema uses names that come from the application domain (e.g., `Statistics`) instead of names that denote solution objects (e.g., `StatisticsVault`). This provides better and more direct traceability to the requirements.
- The transformation of the N-ary association into Java classes focused only on runtime concerns, not on data storage concerns.

10.6.5 Lessons Learned

In the previous section, we mapped the `Statistics` N-ary association class to source code and to a database schema. In doing so, we learned that

- *Applying one transformation leads to new opportunities for other transformations*. By converting the N-ary `Statistics` association into a separate object, we defined a facade shielding control and boundary objects from the details of the `Statistics` representation.
- *Introducing new interfaces results in forwarding existing exceptions*. When introducing a new interface, we need to make sure exceptions are not masked. When we introduced

a facade in this example, we simply forwarded the lower-level exceptions (`UnknownStatistic`) to the `Client` class.

- *Database schema is based on the object design model, not on the source code*. In general, we need to ensure that the analysis concepts are visible in the implementation, providing traceability back to the requirements. Naming classes, attributes, methods, and tables in terms of names used during analysis is critical in maintaining conceptual integrity.

10.7 Further Readings

The history of refactoring can be traced back to program transformation. Many program transformation systems have focused on improving the performance of programs or on generating programs from formal specifications. *Specification and Transformation of Programs* [Partsch, 1990] provides a comprehensive treatment of this topic.

Although program transformation systems are becoming mature and reaching the marketplace, few tools exist for supporting the transformation of object models. *Object-Oriented Modeling and Design* [Rumbaugh et al., 1991] first introduced the concept of object model transformations. Later, *Object-Oriented Modeling and Design for Database Applications* [Blaha & Premerlani, 1998] introduced the concept of transformations of object models to relational database schemas.

Refactoring, made popular by Martin Fowler in *Refactoring: Improving The Design Of Existing Code* [Fowler, 2000] is essentially a variation of program transformation applied to object-oriented programs. Refactorings are applied manually and interleaved with unit tests. Refactoring is one of the cornerstones of Extreme Programming [Beck & Andres, 2005].

10.8 Exercises

10-1 In Web pages, tables consist of rows, which in turn consist of cells. The actual width and height of each cell is computed based in part on its content (e.g., the amount of text in the cell, the size of an image in the cell), and the height of a row is the maximum of the heights of all cells in the row. Consequently, the final layout of a table in a Web page can only be computed once the content of each cell has been retrieved from the Internet. Using the proxy pattern described in Figure 10-7, describe an object model and an algorithm that would enable a Web browser to start displaying a table before the size of all cells is known, possibly redrawing the table as the content of each cell is downloaded.

10-2 Apply the appropriate transformations described in Section 10.4.2 to the associations of Figure 10-29. Assume that all associations are bidirectional and that they can change during the lifetime of each object. Write the source code needed to manage the associations, including class, field, and method declarations, method bodies, and visibility.

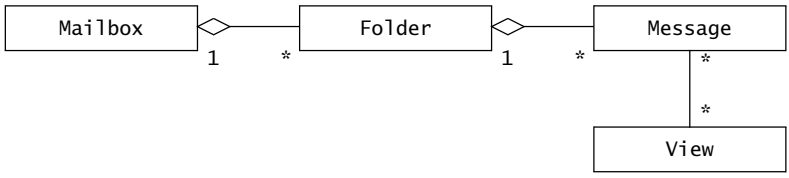


Figure 10-29 Associations among `Messages`, `Folders`, `Mailboxes`, and `Views` in a hypothetical E-mail client (UML class diagram).

10-3 Apply the appropriate transformations described in Section 10.4.2 to the associations of Figure 10-30. Assume that all associations are bidirectional, but that the aggregation associations do not change after each object has been created. In other words, the creator of each class must be modified so that aggregations are initialized during the creation of each object. Write the source code needed to manage the associations, including class, field, and method declarations, method bodies, and visibility.

Figure 10-30 Associations among `League`, `Tournament`, *`Round`*, and `Player` (UML class diagram).

10-4 Figure 10-15 depicts the checking code for the `addPlayer()` method of `Tournament`. Write the checking code for the other constraints associated with `Tournament` depicted in Figure 9-16.

10-5 Write checking code for the contracts of *`TournamentStyle`* and *`Round`* described in Section 9.6.2. Write checking code for preconditions, postconditions, and invariants.

10-6 Design a relational database schema for the object model of Figure 10-30. Assume `Leagues`, `Tournaments`, `Players`, and *`Rounds`* have a `name` attribute and a unique identifier. Additionally, `Tournaments` and *`Rounds`* have `start` and `end` date attributes. When different transformations are available, explain the trade-off involved.

10-7 Draw a class diagram representing the application domain facts below, and map it to a relational schema.

- A project involves a number of participants.
- Participants can take part in a project either as project manager, team leader, or developer.

- Within a project, each developer and team leader is part of at least one team.
- A participant can take part in many projects, possibly in different roles. For example, a participant can be a developer in project A, a team leader in project B, and a project manager in project C. However, the role of a participant within a project does not change.

10-8 There are two general approaches for mapping an association to a set of collections. In Section 10.6.2, we map the N-ary association `Statistics` to two classes, a simple `Statistics` class to store the attributes of the association, and a `StatisticsVault` class to store the state of the links among the association links. In Section 10.4.2, we described an alternative approach where the association links are stored in one or both classes at the ends of the association. In the event associations were stored in both classes, we added mutually recursive methods to ensure that both data structures remained consistent. Use this second approach to map the N-ary `Statistics` association to `Collections`. Discuss the trade-offs you encounter and the relative advantages of each approach.

References

[Beck & Andres, 2005] K. Beck & C. Andres, *Extreme Programming Explained: Embrace Change*, 2nd ed., Addison-Wesley, Reading, MA, 2005.

[Blaha & Premerlani, 1998] M. Blaha & W. Premerlani, *Object-Oriented Modeling and Design for Database Applications*, Prentice Hall, Upper Saddle River, NJ, 1998.

[Date, 2004] C. J. Date, *An Introduction to Database Systems*, 8th ed., Addison-Wesley, Reading, MA, 2004.

[Fowler, 2000] M. Fowler, *Refactoring: Improving The Design of Existing Code*, Addison-Wesley, Reading, MA, 2000.

[Gamma et al., 1994] E. Gamma, R. Helm, R. Johnson, & J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1994.

[Kramer, 1998] R. Kramer, "iContract—The Java design by contract tool," *Technology of Object-Oriented Languages and Systems*, IEEE Computer Society Press, p.295, 1998.

[Partsch, 1990] H. Partsch, *Specification and Transformation of Programs*, Springer-Verlag, 1990.

[Rumbaugh et al., 1991] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, & W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, NJ, 1991.

[Tolkien, 1995] J.R.R. Tolkien, *The Lord of The Rings*, Harper Collins, 1995.