17

Implementation Modeling

Implementation is the final development stage that addresses the specifics of programming languages. Implementation should be straightforward and almost mechanical, because you should have made all the difficult decisions during design. To a large extent, your programming code should simply translate design decisions. You must add details while writing code, but each one should affect only a small part of the program.

17.1 Overview of Implementation

It is now, during implementation, that you finally capitalize on your careful preparation from analysis and design. First you should address implementation issues that transcend languages. This is what we call ***implementation modeling*** and involves the following steps.

- Fine-tune classes. [17.2]
- Fine-tune generalizations. [17.3]
- Realize associations. [17.4]
- Prepare for testing [17.5]

The first two steps are motivated by the theory of transformations. A ***transformation*** is a mapping from the domain of models to the range of models. When modeling, it is important not only to focus on customer requirements, but to also take an abstract mathematical perspective.

17.2 Fine-tuning Classes

Sometimes it is helpful to fine-tune classes before writing code in order to simplify development or to improve performance. Keep in mind that the purpose of implementation is to realize the models from analysis and design. Do not alter the design model unless there is a compelling reason. If there is, consider the following possibilities.

- **Partition a class.** In Figure 17.1, we can represent home and office information for a person with a single class or we can split the information into two classes. Both approaches are correct. If we have much home and office data, it would be better to separate them. If we have a modest amount of data, it may be easier to combine them.

  The partitioning of a class can be complicated by generalization and association. For example, if *Person* was a superclass and we split it into home and office classes, it would be less convenient for the subclasses to obtain both kinds of information. The subclasses would have to multiply inherit, or we would have to introduce an association between the home and office classes. Furthermore, if there were associations to *Person*, you would need to decide how to associate to the partitioned classes.

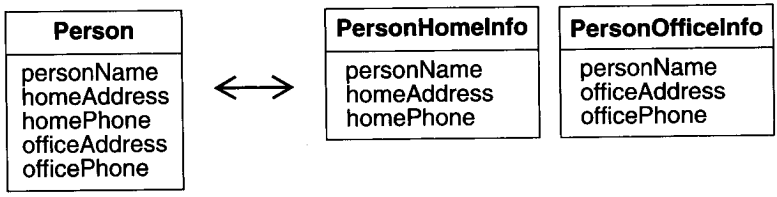

Figure 17.1 Partitioning a class. Sometimes it is helpful to fine-tune a model by partitioning or merging classes.

- **Merge classes.** The converse to partitioning a class is to merge classes. If we had started with *PersonHomeInfo* and *PersonOfficeInfo* in Figure 17.1, we could combine them. Figure 17.2 shows another example with intervening associations. Neither representation is inherently superior, because both are mathematically correct. Once again, you must consider the effects of generalization and association in your decisions.

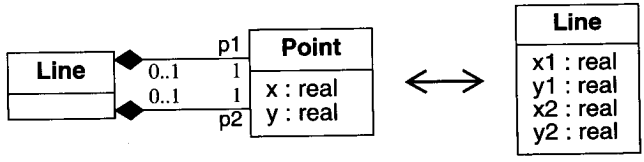

Figure 17.2 Merging classes. It is acceptable to rework your definitions of classes, but only do so for compelling development or performance reasons.

- **Partition / merge attributes.** You can also adjust attributes by partitioning and merging, as Figure 17.3 illustrates.
- **Promote an attribute / demote a class.** As Figure 17.4 shows, we can represent address as an attribute, as one class, or as several related classes. The bottom model would be helpful if we were preloading address data for an application.

ATM example. We may want to split *Customer address* into several classes if we are prepopulating address data. For example, we may preload *city*, *stateProvice*, and *postalCode*

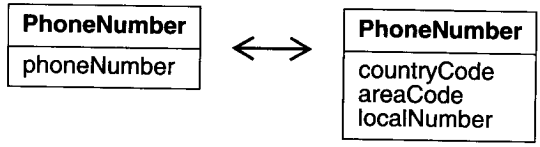

Figure 17.3 Partitioning / merging attributes

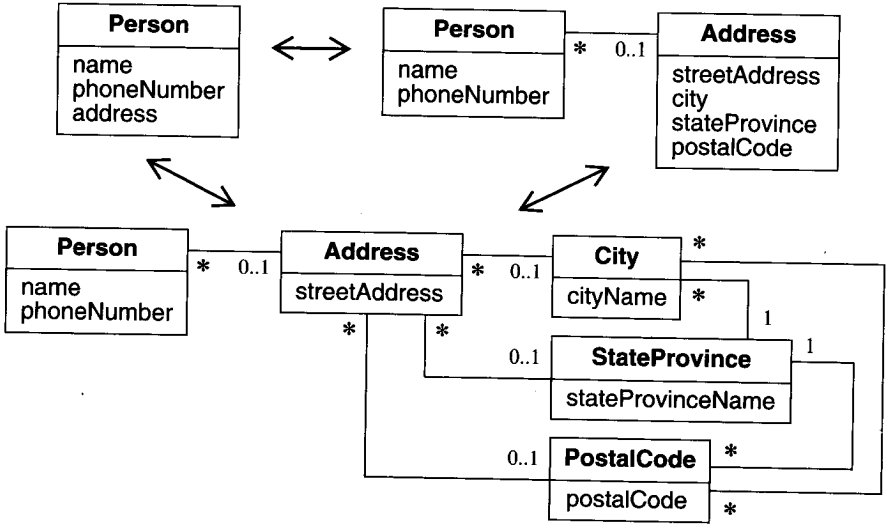

Figure 17.4 Promoting an attribute / demoting a class

data for the convenience of customer service representatives when creating a new *Customer* record.

We may also want to place *Account type* in its own class. Then it would be easier to program special behavior. For example, the screens may look different for checking accounts than for savings accounts.

All in all, the ATM model is small in size and carefully prepared, so we are not inclined to make many changes.

17.3 Fine-tuning Generalizations

As you can reconsider classes, so too you can reconsider generalizations. Sometimes it is helpful to remove a generalization or to add one prior to coding.

Figure 17.5 shows a translation model from one of our recent applications. A language translation service converts a *TranslationConcept* into a *Phrase* in the desired language. A *MajorLanguage* is a language such as English, French, or Japanese. A *MinorLanguage* is a

dialect such as American English, British English, or Australian English. All entries in the application database that must be translated store a *translationConceptID*. The translator first tries to find the phrase for a concept in the specified *MinorLanguage* and then, if that is not found, looks for the concept in the corresponding *MajorLanguage*.

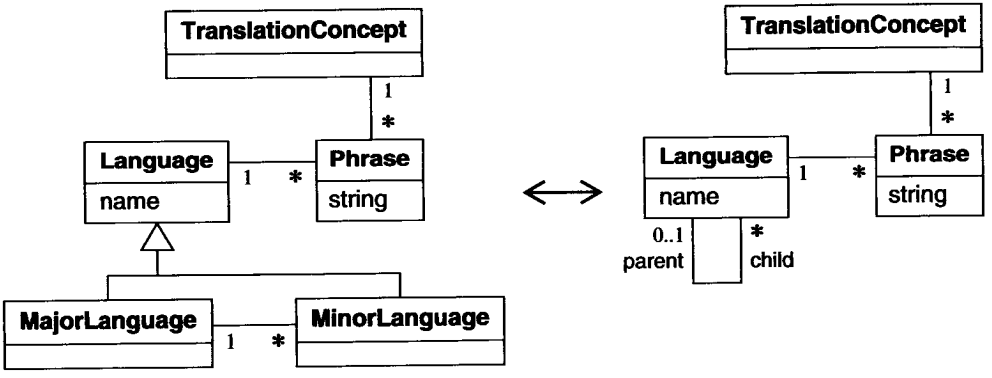

Figure 17.5 Removing / adding generalization. Sometimes it can simplify implementation to remove or add a generalization.

For implementation simplicity, we removed the generalization and used the right model. Since the translation service is separate from the application model, there were no additional generalizations or associations to consider, and it was easy to make the simplification.

ATM example. Back in Section 13.1.1 we mentioned that the ATM domain class model encompassed two applications—ATM and cashier. We did not concern ourselves with this during analysis—the purpose of analysis is to understand business requirements, and the eventual customer does not care how services are structured. Furthermore, we wanted to make sure that both applications had similar behavior. However, now that we are implementing, we must separate the applications and limit the scope to what we will actually build. Figure 17.6 deletes cashier information from the domain class model, leading to a removal of both generalizations.

Figure 17.6 is the full ATM class model. The top half (*Account* and above) presents the domain class model; the bottom half (*UserInterface*, *ConsortiumInterface*, and below) presents the application class model. The operations are representative, but only some are listed.

17.4 Realizing Associations

Associations are the "glue" of the class model, providing access paths between objects. Now we must formulate a strategy for implementing them. Either we can choose a global strategy for implementing all associations uniformly, or we can select a particular technique for each association, taking into account the way the application will use it. We will start by analyzing how associations are traversed.

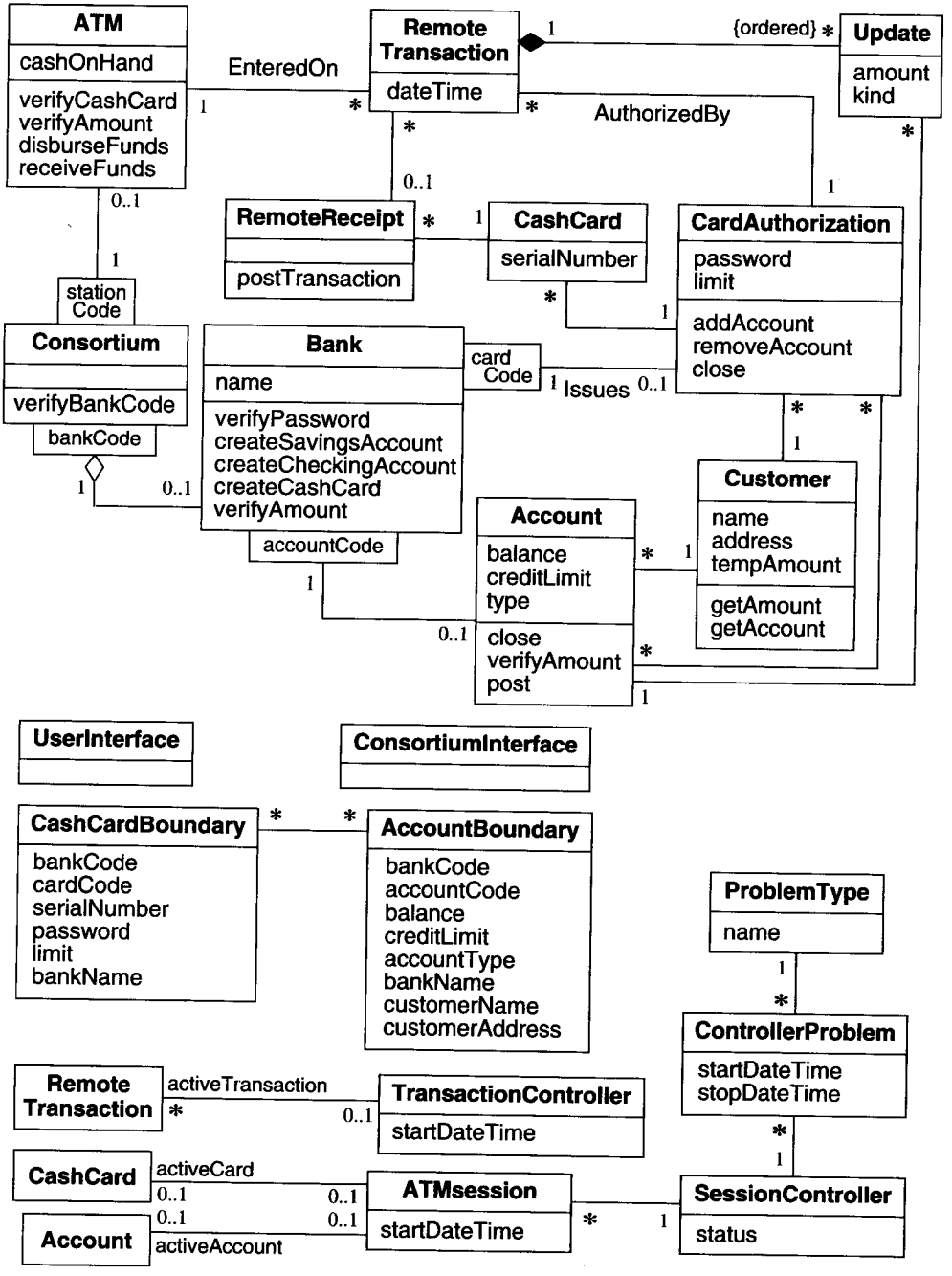

Figure 17.6 Full ATM implementation class model

17.4.1 Analyzing Association Traversal

We have assumed until now that associations are inherently bidirectional, which is certainly true in an abstract sense. But if your application has some associations that are traversed in only one direction, their implementation can be simplified. Be aware, however, that future requirements may change, and you may need to add a new operation later that traverses the association in the reverse direction.

For prototype work we always use bidirectional associations, so that we can add new behavior and modify the application quickly. For production work we optimize some associations. In any case, you should hide the implementation, using access methods to traverse and update the association. Then you can change your decision more easily.

17.4.2 One-way Associations

If an association is traversed only in one direction, you can implement it as a ***pointer***—an attribute that contains an object reference. (Note that this chapter uses the word *pointer* in the logical sense. The actual implementation could be a programming-language pointer, a programming-language reference, or even a database foreign key.) If the multiplicity is "one," as Figure 17.7 shows, then it is a simple pointer; if the multiplicity is "many," then it is a set of pointers.

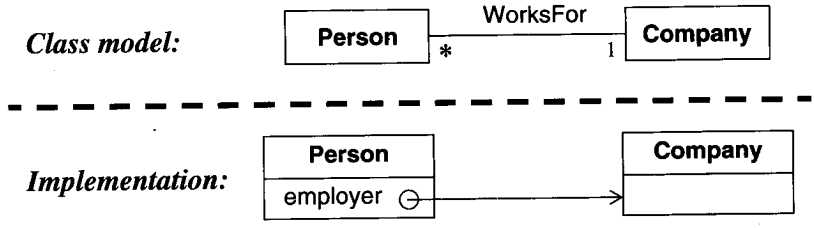

Figure 17.7 Implementing a one-way association with pointers. If an association is traversed only in one direction, you can implement it as a pointer.

17.4.3 Two-way Associations

Many associations are traversed in both directions, although not usually with equal frequency. There are three approaches to their implementation.

- **Implement one-way.** Implement as a pointer in one direction only and perform a search when backward traversal is required. This approach is useful only if there is a great disparity in traversal frequency in the two directions and minimizing both the storage and update costs is important. The rare backward traversal will be expensive.
- **Implement two-way.** Implement with pointers in both directions as Figure 17.8 shows. This approach permits fast access, but if either direction is updated, then the other must also be updated to keep the link consistent. This approach is useful if accesses outnumber updates.

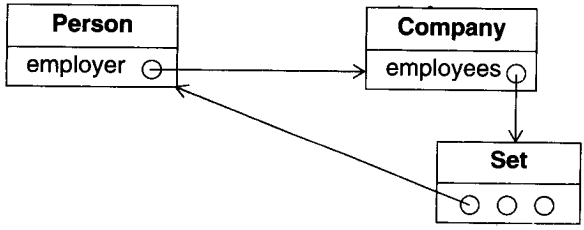

Figure 17.8 Implementing a two-way association with pointers. Dual pointers enable fast traversal of an association in either direction, but introduce redundancy, complicating maintenance.

- **Implement with an association object.** Implement with a distinct association object, independent of either class, as Figure 17.9 shows [Rumbaugh-87]. An association object is a set of pairs of associated objects (triples for qualified associations) stored in a single variable-size object. For efficiency, you can implement an association object using two dictionary objects, one for the forward direction and one for the backward direction. Access is slightly slower than with pointers, but if hashing is used, then access is still constant time. This approach is useful for extending predefined classes from a library that cannot be modified, because the association object does not add any attributes to the original classes. Distinct association objects are also useful for sparse associations, in which most objects of the classes do not participate, because space is used only for actual links.

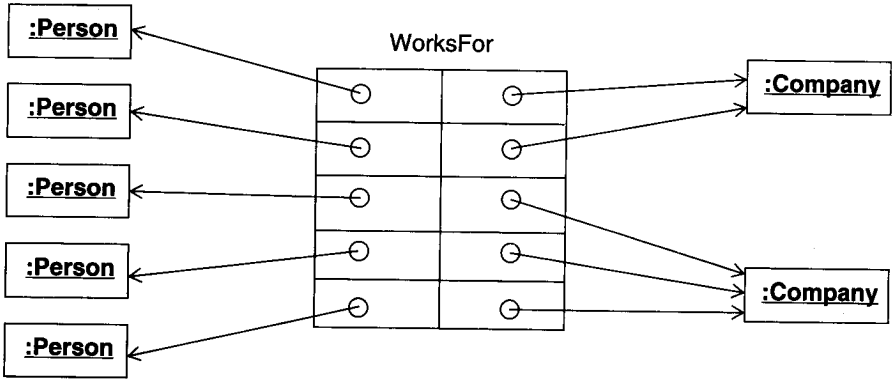

Figure 17.9 Implementing an association as an object. This is the most general approach for implementing associations but requires the most programming skill.

17.4.4 Advanced Associations

The appropriate techniques for implementing advanced associations vary.

- **Association classes**. The usual approach is to promote the association to a class. This handles any attributes of the association as well as associations of the association class. Note that promotion changes the meaning of the model; the promoted association has identity of its own, and methods must compensate to enforce the dependency of the association class on the constituent classes.

  Alternatively, if the association is one-to-one and has no further associations, you can implement the association with pointers and store any attributes for the association as attributes of either class. Similarly, if the association is one-to-many and has no further associations, you can implement the association with pointers and store attributes for the association as attributes of the "many" class, since each "many" class appears only once for the association.
- **Ordered associations**. Use an ordered set of pointers similar to Figure 17.8 or a dictionary with an ordered set of pairs similar to Figure 17.9.
- **Sequences**. Same as ordered association, but use a list of pointers.
- **Bags**. Same as ordered association, but use an array of pointers.
- **Qualified associations**. Implement a qualified association with multiplicity "one" as a dictionary object. using the techniques of Figure 17.9. Qualified associations with multiplicity "many" are rare, but you can implement them as a dictionary of object sets.
- **N-ary associations**. Promote the association to a class. Note that there is a change in identity and that you must compensate with additional programming, similar to that for association classes.
- **Aggregation**. Treat aggregation like an ordinary association.
- **Composition**. You can treat composition like an ordinary association. You will need to do some additional programming to enforce the dependency of the part on the assembly.

17.4.5 ATM Example

Exercise 17.1 addresses the implementation of associations from the ATM model.

17.5 Testing

If you have carefully modeled your application, as we advise, you will reduce errors in your software and need less testing. Nevertheless, testing is still important. Testing is a quality-assurance mechanism for catching residual errors. Furthermore, testing provides an independent measure of the quality of your software. The number of bugs found for a given testing effort is an indicator of software quality, and you should find fewer bugs as you become proficient at modeling. You should keep careful records of the bugs that you find, as well as customer complaints.

If your software is sound, the primary difficulty for developers is in finding the occasional, odd error. Fixing the errors is a much easier problem. (In contrast, if your software is haphazard, it can also be difficult to fix the errors.)

You need to test at every stage of development, not just during implementation. The nature of the testing changes, however, as you proceed. During analysis, you test the model against user expectations by asking questions and seeing if the model answers them. During design, you test the architecture and can simulate its performance. During implementation, you test the actual code—the model serves as a guide for paths to traverse.

Testing should progress from small pieces to ultimately the entire application. Developers should begin by testing their own code, their classes and methods—this is called ***unit testing***. The next step is ***integration testing***—that is, how the classes and methods fit together. You do integration testing in successive waves, putting code together in increasing chunks of scope and behavior. It is important to do integration testing early and often to ensure that the pieces of code cleanly fit together (see Chapter 21). The final step is ***system testing***, where you check the entire application.

17.5.1 Unit Testing

Developers normally check their own code and do the unit and integration testing, because they understand the detailed logic and likely sources of error. Unit testing follows the same principles as in pre-OO days: developers should try to cover all paths and cases, use special values of arguments, and try extreme and "off-by-one" values for arguments. If your methods and classes are simple and focused, it will be easier to prepare unit tests.

It is a good idea to instrument objects and methods. You can place assertions (preconditions, postconditions, invariants) in your code to trap errors. You should try to detect problems near the source (where they are easier to understand) rather than downstream (where they can be confusing).

We agree with the use of paired programmers and aggressive code inspection that is part of the agile programming movement. Along the same lines, we also recommend formal software reviews (see Chapter 22), where developers present their work to others and receive comments.

17.5.2 System Testing

Ideally, a separate team apart from the developers should carry out system testing—this is a natural role for a quality assurance (QA) organization. QA should derive their testing from the analysis model of the original requirements and prepare their test suite in parallel to other development activities. Then the system testers are not distracted by the details of development and can provide an independent assessment of an application, reducing the chance of oversights. Once alpha testing is complete, customers perform beta tests, and then if the software looks good, it is ready for general release.

The scenarios of the interaction model define system-level test cases. You can generate additional scenarios from the use cases or state machines. Pick some typical test cases, but also consider atypical situations: zero iterations, the maximum number of iterations, coincidence of events if permitted by the model, and so on. Strange paths though the state machine make good test cases, because they check default assumptions. Also pay attention to performance and stress the software with multiuser and distributed access, if that is appropriate.

As much as possible, use a test suite. The test suite is helpful for rechecking code after bug fixes and detecting errors that creep into future software releases. It can be difficult to automate testing when you have an application with an interactive user interface, but even then you can still document your test scripts for later use.

ATM example. We have carefully and methodically prepared the ATM model. Consequently we would be in a good position for testing, if we were to build a production application.

17.6 Chapter Summary

Implementation is the final development stage that addresses the specifics of programming languages. First you should address implementation issues that transcend languages—we call this implementation modeling. Sometimes it is helpful to fine-tune classes and generalizations before writing code in order to simplify development or to improve performance. Do this only if you have a compelling reason.

Associations are a key concept in UML class modeling, but are poorly supported by most programming languages. Nevertheless, you should keep your thinking clear by using associations as you study requirements and then necessarily degrade them once you reach implementation. There are two primary ways of implementing associations with programming languages—with pointers (for one or both directions) or with association objects. An association object is a pair of dictionary objects, one for the forward direction and one for the backward direction.

Even though careful modeling reduces errors, it does not eliminate the need for testing. You will need unit, integration, and system tests. For unit testing, developers check the classes and methods of their own code. Integration testing combines multiple classes and methods and subjects them to additional tests. System testing exercises the overall application and ensures that it actually delivers the requirements originally uncovered during analysis.

association object	implementation modeling	transformation
association traversal	integration testing	two-way association
dictionary object	one-way association	unit testing
fine-tuning classes	pointer	
fine-tuning generalizations	system testing	

Figure 17.10 Key concepts for Chapter 17

Bibliographic Notes

Transformations provide the motivation for Section 17.2 and Section 17.3. [Batini-92] presents a comprehensive list of transformations. [Blaha-96] and [Blaha-98] present additional transformations.