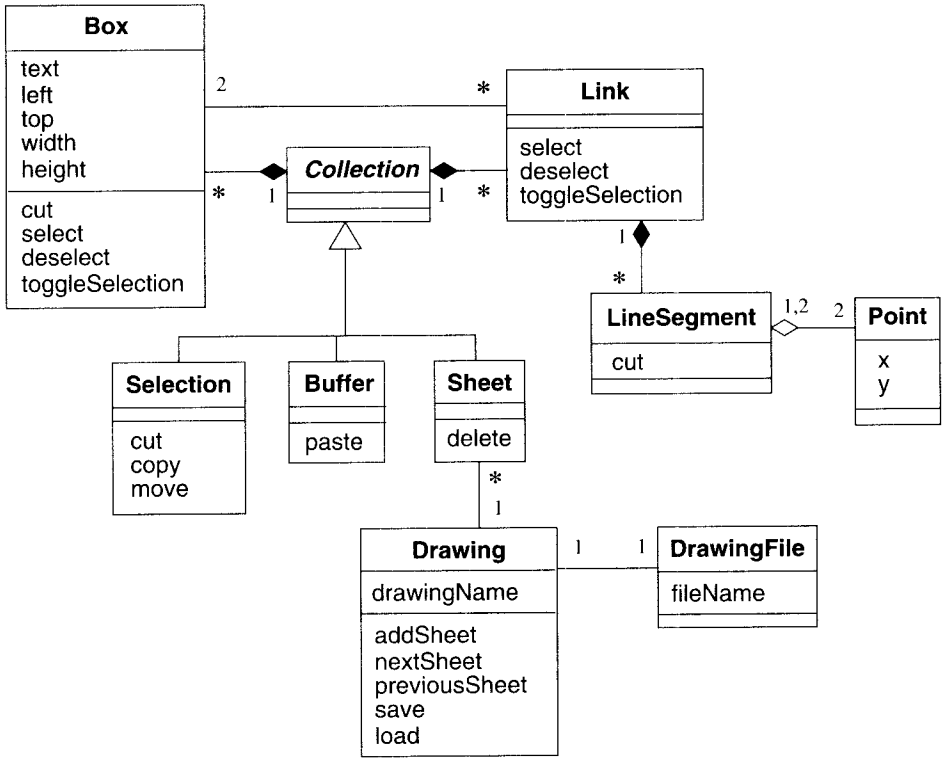


Figure E18.4 Class diagram for a diagram editor

e. Given two boxes, determine all links between them.
f. Given a selection and a sheet, determine which links connect a selected box to a deselected box.
g. Given two boxes and a link, produce an ordered set of points. The first point is where the link connects to the first box, the last point is where the link connects to the second box, and intermediate points trace the link.

19

Databases

The OO paradigm is versatile and applies to databases as well as programming code. It might surprise you, but you can implement UML models not only with OO databases but also with relational databases. The resulting databases are efficient, coherent, and extensible.

You prepare a database by first performing the analysis steps described in Chapter 12 and constructing a domain model. The remaining methodology chapters in Part 2 apply mostly to design of programming code and to a lesser extent to databases. This chapter resumes where Chapter 12 ends. How can we map a model to database structures and tune the result for fast performance? How can we couple the resulting database to programming code? This chapter also includes a brief introduction to databases for new readers.

The chapter emphasizes relational databases because they dominate the marketplace. OO databases are practical only for niche applications and are discussed at the end of the chapter.

19.1 Introduction

19.1.1 Database Concepts

A ***database*** is a permanent, self-descriptive store of data that is contained in one or more files. Self-description is what sets a database apart from ordinary files. A database contains the data structure or ***schema***—description of data— as well as the data.

A ***database management system (DBMS)*** is the software for managing access to a database. One major objective of OO technology is to promote software reuse; for data-intensive applications DBMSs can replace much application code. You are achieving reuse when you use generic DBMS code, rather than custom-written application code. There are additional reasons for using a DBMS.

- **Data protection**. DBMSs protect data from accidental loss due to hardware crashes, disk media failures, and application errors.
- **Efficiency**. DBMSs have efficient algorithms for managing large quantities of data.
- **Sharing between users**. Multiple users can access the database at the same time.
- **Sharing between applications**. Multiple application programs (presumably related) can read and write data to the same database. A database is a neutral medium that promotes communication among programs.
- **Data quality**. You can specify rules that data must satisfy. A DBMS can control the quality of its data over and above facilities that application programs may provide.
- **Data distribution**. You can partition data across various sites, organizations, and hardware platforms. The DBMS keeps the fragmented data consistent.
- **Security**. A DBMS can restrict reading and writing of data to authorized users.

19.1.2 Relational Database Concepts

A ***relational database*** has data that is perceived as tables. A ***relational DBMS (RDBMS)*** manages tables of data and associated structures that increase the functionality and performance of tables. RDBMSs have benefitted from a clear definition by an authoritative figure (EF Codd, the inventor of relational databases) and a standard access language (SQL [Melton-93]). All RDBMSs support a common core of SQL for defining tables, manipulating data in tables, and controlling access to tables. Variations exist for data types, performance tuning, programming access, and system data, though the standard is gradually subsuming these areas. An RDBMS has three major aspects: data structure, operators, and constraints.

- **Data structure**. A relational database appears as a collection of tables. Tables have a specific number of columns and an arbitrary number of rows with a value stored at each row-column intersection. Figure 19.1 shows two sample tables. The *Person* table has five columns and four rows; the *Company* table has three columns and three rows. The column names in boldface are primary keys (to be explained). Note that Jane Brown lacks an employer. ***Null*** means that an attribute value is unknown or not applicable for a given row.

  RDBMSs use special techniques—such as indexing, hashing, and sorting—to speed access, because literal tables are much too slow for practical needs. These tuning techniques are transparent and not visible in the commands for reading and writing to tables. The RDBMS decides when tuning structures are helpful in processing a query, and if so, automatically uses them. The RDBMS automatically updates tuning structures whenever the corresponding tables are modified.
- **Operators**. SQL provides operators for manipulating tables. The SQL *select* statement reads the data in tables. The syntax looks something like (keywords are capitalized):

```
SELECT columnList
FROM tableList
WHERE predicateIsTrue
```

Person table

personID	lastName	firstName	address	employer
1	Smith	Jim	314 Olive St.	1001
5	Brown	Moe	722 Short St.	1002
999	Smith	Jim	1561 Main Dr.	1001
14	Brown	Jane	722 Short St.	NULL

Company table

companyID	companyName	address
1001	Ajax Widgets	33 Industrial Dr.
1002	AAA liquors	724 Short St.
1003	Win-more Sports	1877 Broadway

Figure 19.1 Sample tables. A relational DBMS presents data as tables.

Logically (the actual implementation is more efficient) the RDBMS combines the various tables into one temporary table. The column list specifies the columns to retain. The predicate specifies the rows to retain. The RDBMS returns the resulting data as the answer to the query. SQL has additional commands for inserting, deleting, and updating data in tables.

Interactive SQL commands are set-oriented; they operate on entire tables rather than individual rows or values. SQL provides a similar language for use with application programs that has a row-at-a-time interface.

- **Constraints.** An RDBMS can enforce many constraints (such as candidate, primary, and foreign keys) that are defined as part of the database structure. An RDBMS refuses to store data that violates constraints, returning an error to the user or requesting program.

  A ***candidate key*** is a combination of columns that uniquely identifies each row in a table. The combination must be minimal and include only those columns that are needed for unique identification. No column in a candidate key can be null.

  A ***primary key*** is a candidate key that is preferentially used to access the records in a table. A table can have at most one primary key; normally each table should have a primary key. The boldface in Figure 19.1 indicates the primary key of each table.

  A ***foreign key*** is a reference to a candidate key (normally a reference to a primary key) and is the glue that binds tables. In Figure 19.1 *employer* is a foreign key in the *Person* table that refers to *companyID* in the *Company* table. It would not be permissible to change Moe Brown's *employer* to 1004, since the *Company* table does not define 1004. If the row for Ajax Widgets were deleted in the *Company* table, then both Jim Smith rows would have to be deleted or have their *employer* set to null. The foreign-to-primary-key binding forms a frequent navigation path between tables.

19.1.3 Normal Forms

A ***normal form*** is a guideline for relational database tables that increases data consistency. As tables satisfy higher levels of normal forms, they are less likely to store redundant or contradictory data. Developers can violate normal forms for good cause, such as to increase performance for a database that is read and seldom updated. Such a relaxation is called ***denormalization***. The important issue with normal forms is to violate them deliberately and only when necessary.

Normal forms were first used in the 1970s and 1980s. At that time, developers built databases by creating a list of desired fields, which they then had to organize into meaningful groups before storing them in a database. That was the purpose of normal forms. Normal forms organize fields into groups according to dependencies between fields. Unfortunately, it is easy to overlook dependencies. If any are missed, the resulting database structure may be flawed.

UML models provide a better way to prepare databases. Instead of focusing on the fine granularity of fields, developers think in terms of groups of fields—that is, classes. UML models do not diminish the validity of normal forms—normal forms apply regardless of the development approach.

However, UML modeling does eliminate the need to check normal forms. If developers build a sound model, it will intrinsically satisfy normal forms. The converse also holds—a poor model is unlikely to satisfy normal forms. Furthermore, if developers cannot build a sound model, they will probably be unable to find all the dependencies that are required for checking normal forms. It is less difficult to build models than to find all the dependencies.

The bottom line is that developers can still check normal forms if they want to after modeling, but such a check is unnecessary.

19.1.4 Choosing a DBMS Product

In order to build an application, you must choose a specific DBMS product. Because the core features have been set by the SQL standard, you should choose an RDBMS vendor according to pragmatic concerns.

- **Market share**. Oracle, IBM, and Microsoft are the major market players. The staying power of other vendors is less clear. You may also want to consider an open-source DBMS such as MySQL or PostgreSQL.
- **Vendor and third-party support**. DBMSs are a big commitment for an organization and require ongoing help.
- **Other applications**. You reduce administrative and license costs if you use the same vendor or a small number of vendors for your applications.

With each new product release the major vendors tend to jump ahead of the competition's features and performance benchmarks, only to be surpassed themselves when a competitor has its own new release. Over the long term there tends to be little difference in features and performance, so you should not dwell on them when choosing a product.

19.2 Abbreviated ATM Model

Figure 19.2 shows the portion of the ATM model that this chapter will use as an example. We added *CheckingAccount* and *SavingsAccount* so that we can discuss generalization. We also added the *Address* class to assist our explanation.

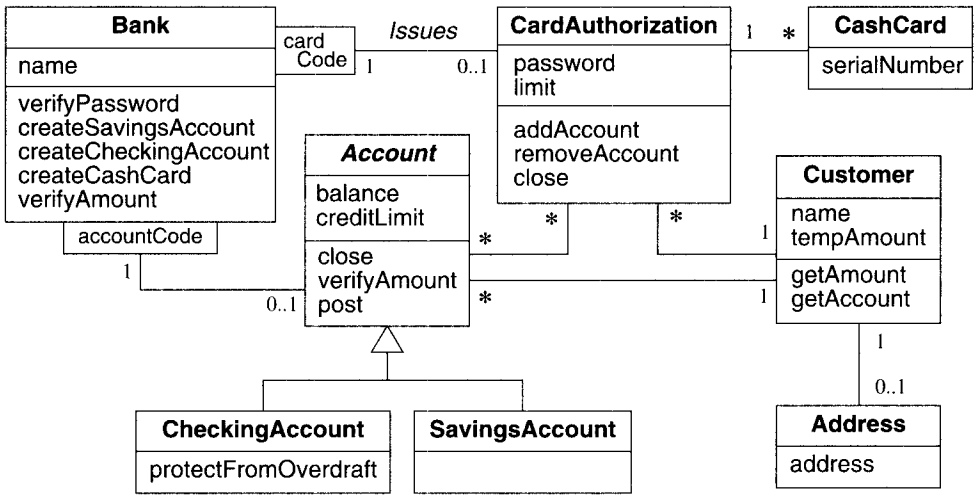


Figure 19.2 Abbreviated ATM implementation class model used in this chapter

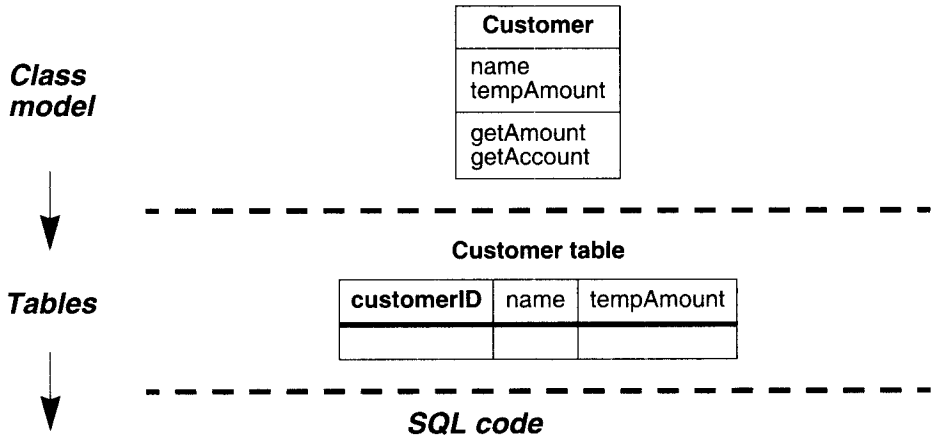
19.3 Implementing Structure—Basic

You can readily translate the class model into SQL code. Many tools will do this, but you should still know the rules. Then you understand what the tools are doing and can spot check their results. RDBMSs provide good support for classes and associations but lack support for inheritance, so you must use a workaround. You should perform the following initial tasks.

- Implement classes. [19.3.1]
- Implement associations. [19.3.2]
- Implement generalizations. [19.3.3]
- Implement identity [19.3.4]

19.3.1 Classes

Normally you should map each class to a table and each attribute to a column (Figure 19.3). You can add columns for an object identifier and associations (to be explained). The boldface indicates the primary key. Keywords are in uppercase. Note that operations do not affect table structure. We have chosen data types and *not-null* constraints that seem appropriate for the




```
CREATE TABLE Customer
( customer_ID NUMBER(30)    CONSTRAINT nn_customer1 NOT NULL,
  cust_name   VARCHAR2(50) CONSTRAINT nn_customer2 NOT NULL,
  temp_amount NUMBER(12,2),
CONSTRAINT pk_customer PRIMARY KEY (customer_ID));
```

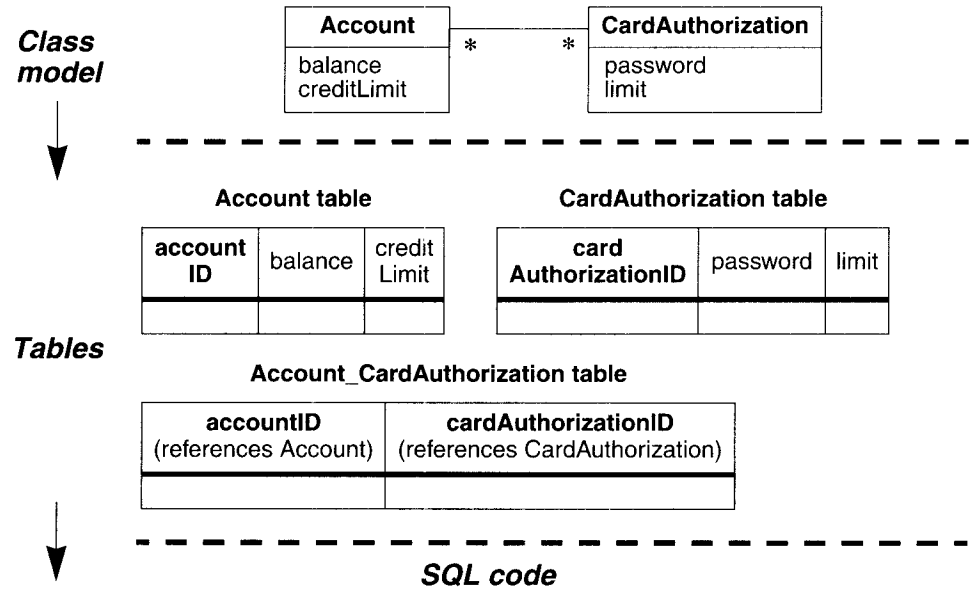
Figure 19.3 Implementing classes. Make each class a table.

problem. All our examples use Oracle syntax. The *nn_customer1* and *nn_customer2* are names of the not-null constraints. The *pk_customer* is the name of the primary key constraint.

19.3.2 Associations

The implementation rules for associations depend on the multiplicity.

- **Many-to-many associations**. Implement the association with a table and make the association's primary key the combination of the classes' primary keys (Figure 19.4). If the association has attributes, they become additional columns.
- **One-to-many associations**. Each one becomes a foreign key buried in the table for the "many" class (Figure 19.5). If there had been a name on the "one" end of the association, we would have used it as the foreign key name. We presume that *serialNumber* is unique for *CashCard*.
- **One-to-one associations**. These seldom occur. You can handle them by burying a foreign key in either class table (Figure 19.6).
- **N-ary associations**. They also seldom occur. You can treat them like many-to-many associations and create a table for the association. Typically, the primary key of the n-ary association combines the primary keys of the related tables.
- **Association classes**. An ***association class*** is an association that is also a class. It is easier to establish the proper dependencies if you make each association class into a table, regardless of the multiplicity.



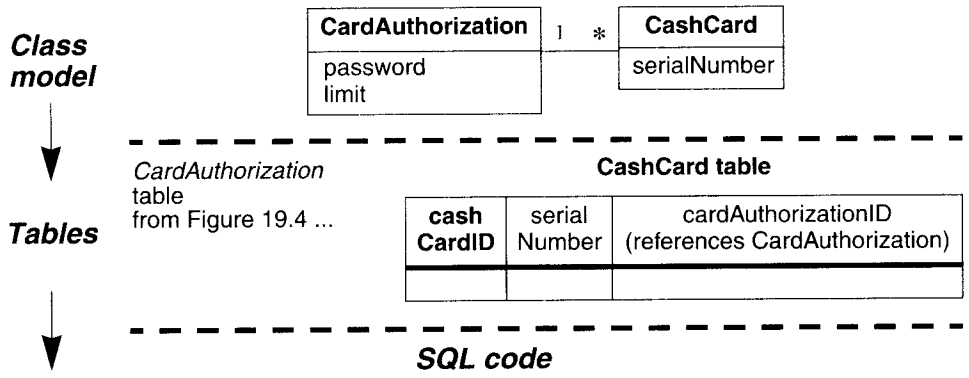

```
CREATE TABLE Account
( account_ID    NUMBER(30)    CONSTRAINT nn_account1 NOT NULL,
  balance       NUMBER(12,2) CONSTRAINT nn_account2 NOT NULL,
  credit_limit NUMBER(12,2),
CONSTRAINT pk_account PRIMARY KEY (account_ID));

CREATE TABLE Card_Authorization
( card_auth_ID NUMBER(30) CONSTRAINT nn_cardauth1 NOT NULL,
  password      VARCHAR2(50),
  limit         NUMBER(12,2),
CONSTRAINT pk_cardauth PRIMARY KEY (card_auth_ID));

CREATE TABLE Acct_CardAuth
( account_ID    NUMBER(30) CONSTRAINT nn_acctca1 NOT NULL,
  card_auth_ID NUMBER(30) CONSTRAINT nn_acctca2 NOT NULL,
CONSTRAINT pk_acctca PRIMARY KEY (account_ID, card_auth_ID));
```

Figure 19.4 Implementing many-to-many associations. Make each one a table.

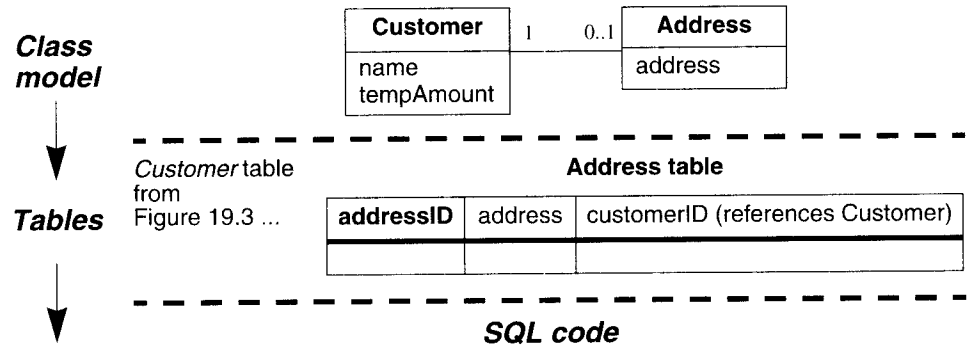
- **Qualified associations**. Qualified associations follow the same rules as the underlying association without the qualifier. Thus we treat Figure 19.7 like a one-to-many association (a bank has many accounts). The notation *ckn* (*n* is a number) denotes a candidate key. Many qualified associations have a candidate key involving the qualifier.
- **Aggregation, composition**. Aggregation and composition follow the same implementation rules as association.




```
Card_Authorization code from Figure 19.4 ...

CREATE TABLE Cash_Card
( cash_card_ID NUMBER(30)    CONSTRAINT nn_cashcard1 NOT NULL,
  serial_num   VARCHAR2(50) CONSTRAINT nn_cashcard2 NOT NULL,
  card_auth_ID NUMBER(30)    CONSTRAINT nn_cashcard3 NOT NULL,
CONSTRAINT pk_cashcard PRIMARY KEY (cash_card_ID),
CONSTRAINT uq_cashcard1 UNIQUE (serial_num));
```

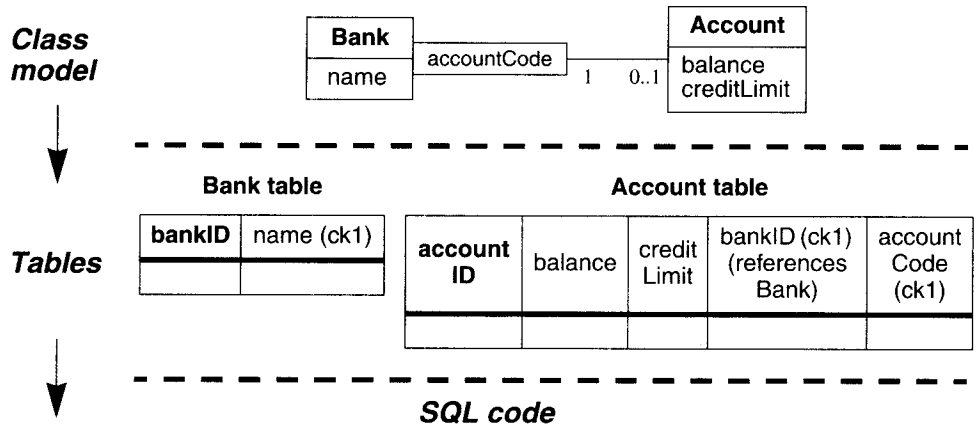
Figure 19.5 Implementing one-to-many associations. Bury each one as a foreign key in the "many" class table.




```
Customer code from Figure 19.3 ...

CREATE TABLE Address
( address_ID  NUMBER(30)     CONSTRAINT nn_address1 NOT NULL,
  address     VARCHAR2(200) CONSTRAINT nn_address2 NOT NULL,
  customer_ID NUMBER(30)     CONSTRAINT nn_address3 NOT NULL,
CONSTRAINT pk_address PRIMARY KEY (address_ID));
```

Figure 19.6 Implementing one-to-one associations. Bury a foreign key in either class table.




```
CREATE TABLE Bank
( bank_ID       NUMBER(30)    CONSTRAINT nn_bank1 NOT NULL,
  bank_name     VARCHAR2(50) CONSTRAINT nn_bank2 NOT NULL,
CONSTRAINT pk_bank PRIMARY KEY (bank_ID),
CONSTRAINT uq_bank1 UNIQUE (bank_name));

CREATE TABLE Account
( account_ID    NUMBER(30)    CONSTRAINT nn_account1 NOT NULL,
  balance       NUMBER(12,2) CONSTRAINT nn_account2 NOT NULL,
  credit_limit NUMBER(12,2),
  bank_ID       NUMBER(30)    CONSTRAINT nn_account3 NOT NULL,
  account_code VARCHAR2(50) CONSTRAINT nn_account4 NOT NULL,
CONSTRAINT pk_account PRIMARY KEY (account_ID),
CONSTRAINT uq_account1 UNIQUE (bank_ID, account_code));
```

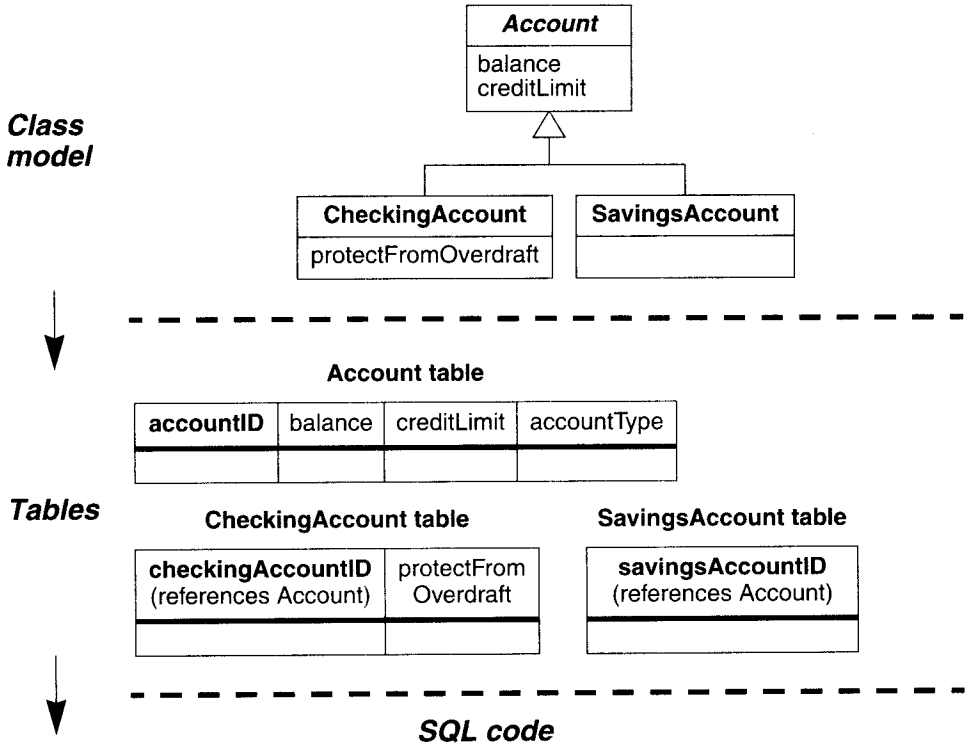
Figure 19.7 Implementing qualified associations. Treat each one like the association without the qualifier.

19.3.3 Generalizations

The implementation rules for generalization depend on whether there is single or multiple inheritance.

- **Single inheritance.** The simplest approach is to map the superclass and subclasses each to a table, as Figure 19.8 shows. The generalization set name (*accountType*) indicates the appropriate subclass table for each superclass record. For a multilevel generalization, you apply the mappings one level at a time.

  In the figure, note that the primary key names vary, but an object should have the same primary key value throughout an inheritance hierarchy. Thus "Joe's checking account" may have one row in the *Account* table with *account_ID* 101 and another row in




```
CREATE TABLE Account
( account_ID    NUMBER(30)    CONSTRAINT nn_account1 NOT NULL,
  balance       NUMBER(12,2)  CONSTRAINT nn_account2 NOT NULL,
  credit_limit  NUMBER(12,2),
  account_type  VARCHAR2(20)  CONSTRAINT nn_account3 NOT NULL,
CONSTRAINT pk_account PRIMARY KEY (account_ID));

CREATE TABLE Checking_Account
( chk_acct_ID   NUMBER(30)   CONSTRAINT nn_chkacct1 NOT NULL,
  protect_odrft VARCHAR2(1)  CONSTRAINT nn_chkacct2 NOT NULL,
CONSTRAINT pk_chkacct PRIMARY KEY (chk_acct_ID));

CREATE TABLE Savings_Account
( sav_acct_ID   NUMBER(30) CONSTRAINT nn_savacct1 NOT NULL,
CONSTRAINT pk_savacct PRIMARY KEY (sav_acct_ID));
```

Figure 19.8 Implementing generalizations (single inheritance). Map the superclass and subclasses each to a table.

the *Checking_Account* table with *chk_acct_ID* 101. We prefer to tie ID names to class names (*account_ID*, *chk_acct_ID*, and *sav_acct_ID*), rather than use the same name (*account_ID*) for all the tables that implement a generalization—this makes it easier to handle multilevel generalizations.

Note that you should not eliminate subclasses that have no attributes, such as *SavingsAccount* in Figure 19.8. Such a performance optimization is seldom important and it complicates the enforcement of foreign key dependencies (Section 19.4.1).

- **Multiple inheritance**. You can handle multiple inheritance from disjoint classes with separate superclass and subclass tables. For multiple inheritance from overlapping classes, you should use one table for each superclass, one table for each subclass, and one table for the generalization.

19.3.4 Identity

Aside from special situations, such as temporary tables, every table should have a primary key. Without any explanation, we have been using object identity. This is our preferred approach, but we should mention another approach that is also common in the database literature. Figure 19.9 shows the two options.

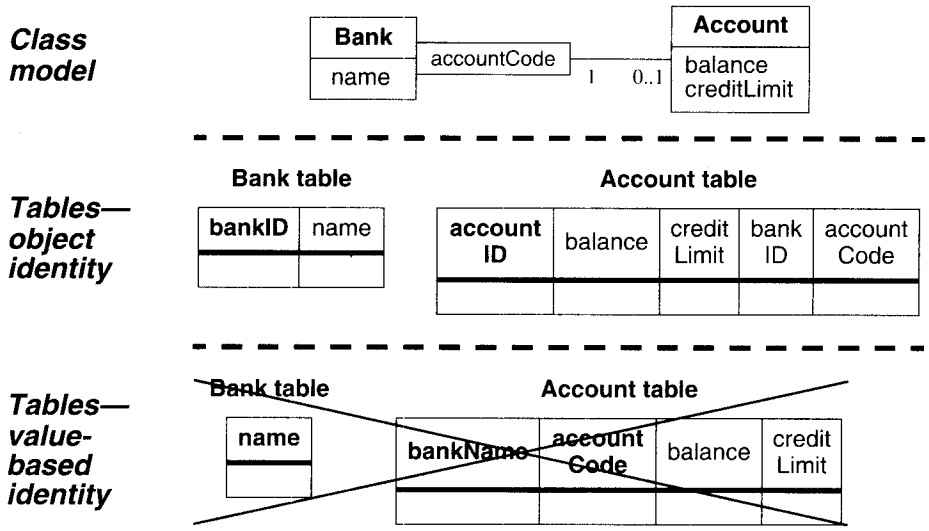


Figure 19.9 Object vs. value-based identity. We recommend the use of object identity.

- **Object identity**. Add an artificial number attribute (an object ID) to each class table and make it the primary key. The primary key for each association table consists of identifiers from the related classes.

  Object identifiers have the advantage of being a single attribute, small, and uniform in size. Most RDBMSs can efficiently allocate identifiers. However, object identifiers

can make a database more difficult to read during debugging and maintenance. IDs also complicate database merges; ID values may contend and need to be reassigned. You should not display artificial numbers to users.

- **Value-based identity**. Identify each object with some combination of real-world attributes. The primary key for each association table consists of primary keys from the related classes.

  Value-based identity has different trade-offs. Primary keys have intrinsic meaning, making it easier to debug the database. On the downside, value-based primary keys can be difficult to change. A change may propagate to other tables. Some classes do not have natural real-world identifiers.

We recommend that you use object identity for RDBMS applications. The resulting uniformity and simplicity outweighs any additional debugging effort. Furthermore, object identity is more consistent with the spirit of object orientation—that objects have intrinsic identity apart from their properties. OO languages implement identity with pointers or look-up tables into pointers; an ID is the equivalent database construct.

19.3.5 Summary of Basic Rules for RDBMS Implementation

Table 19.1 summarizes the basic rules for implementing RDBMS structure.

Concept	UML construct	Recommended implementation rule
Class	Class	Map each class to a table and each attribute to a column in the table
Association (End names become foreign key names.)	Many-to-many	Use distinct table
	One-to-many	Use buried foreign key
	One-to-one	
	N-ary	Use distinct table
	Association class	
	Qualified	Same rules as underlying nonqualified association
Aggregation	Aggregation	Same rules as association
	Composition	
Generalization	Single inheritance	Create separate tables for the superclass and each subclass
	Disjoint multiple inheritance	
	Overlapping multiple inheritance	Same as disjoint multiple inheritance + generalization table to bind superclass and subclass records

Table 19.1 Summary of basic rules for implementing relational databases. These rules are embedded in most database generation tools.

19.4 Implementing Structure—Advanced

The prior section explained how to define tables for each construct in the UML class model. Now we cover advanced aspects that boost performance and ensure data quality. You should perform the following additional tasks.

- Implement foreign keys. [19.4.1]
- Implement check constraints. [19.4.2]
- Implement indexes. [19.4.3]
- Consider views. [19.4.4]

19.4.1 Foreign Keys

Foreign keys arise from generalizations and associations. When a foreign key is defined, the RDBMS *guarantees* that there will be no dangling references—the RDBMS refuses to perform any updates that would cause a dangling reference. Since an application knows that defined foreign keys will always be intact, it need not include error checking for them.

In addition, many RDBMSs can propagate the effects of deletions and updates that affect foreign keys. When you use object identity, as we have suggested, there is no need to propagate updates—IDs are invariant and never change. However, it is helpful to declare the response to deletions, as we will now explain.

For generalizations you should specify *on delete cascade* for each subclass table. Figure 19.10 shows the foreign-key statements that we would add to Figure 19.8. Recall that generalization structures the description of an object, with each level providing part of the description. An application must combine superclass and subclass records to reconstitute an entire object. Accordingly, with *on delete cascade*, deletion of a superclass record causes the deletion of the corresponding subclass record, and this deletion propagates downward for each generalization level.

```
ALTER TABLE Checking_Account ADD CONSTRAINT fk_chkacct1
  FOREIGN KEY chk_acct_ID
  REFERENCES Account ON DELETE CASCADE;

ALTER TABLE Savings_Account ADD CONSTRAINT fk_savacct1
  FOREIGN KEY sav_acct_ID
  REFERENCES Account ON DELETE CASCADE;
```

Figure 19.10 Maintaining foreign keys for generalization. Each subclass table should specify *on delete cascade* for the foreign key to the superclass.

In a similar manner we would like to propagate deletion of a subclass record upward to the superclass, but SQL unfortunately does not support this reverse direction. To compensate, you must write programming code to propagate deletions up a generalization hierarchy.

You should also declare foreign keys to enforce associations, as Figure 19.11 shows. The appropriate deletion action—cascade or no action—depends on the model's meaning. For

example, we might want the deletion of a customer record to cause the deletion of the corresponding address record—this is *on delete cascade* (the first statement in Figure 19.11). Alternatively, you might want to prevent the deletion of a customer who has accounts (to avoid accidental loss of extensive data). The user must first delete all the accounts and only then can delete the customer. In the SQL standard you prevent deletion by specifying *on delete no action*; the Oracle equivalent is to omit a delete action (the second statement in Figure 19.11).

```
ALTER TABLE Address ADD CONSTRAINT fk_address1
  FOREIGN KEY customer_ID
  REFERENCES Customer ON DELETE CASCADE;

ALTER TABLE Account ADD CONSTRAINT fk_account2
  FOREIGN KEY customer_ID
  REFERENCES Customer;
```

Figure 19.11 Maintaining foreign keys for association. There are two possibilities, which depend on the model's meaning.

19.4.2 Check Constraints

SQL also has general constraints that can enforce the values of an enumeration. Such enforcement is especially helpful for implementing a generalization set name. Figure 19.12 adds a check constraint to the example of Figure 19.8.

```
ALTER TABLE Account ADD CONSTRAINT enum_account1
  CHECK (account_type IN ('Checking_Account',
  'Savings_Account'));
```

Figure 19.12 Enforcing a generalization set name. An SQL check constraint can enforce the values of an enumeration.

19.4.3 Indexes

Most RDBMSs create indexes as a side effect of SQL primary key and candidate key (*unique*) constraints. (An ***index*** is a data structure that maps column values into database table rows.) You should also create an index for every foreign key that is not covered by a primary key or candidate key constraint. For example, the primary key for *Acct_CardAuth* (Figure 19.4) causes an RDBMS to build an index that ensures fast access to *account_ID* as well as *account_ID* + *card_auth_ID*. An additional index on *card_auth_ID* (Figure 19.13) ensures that this field accessed alone is also fast.

These indexes are *critically* important. Foreign-key indexes enable quick combination of tables. A lack of indexes can cause RDBMS performance to degrade by orders of magni-

```
CREATE INDEX index_acctca1 ON Acct_CardAuth (card_auth_ID);
```

Figure 19.13 Defining indexes. Every foreign key should be covered by an index.

tude. Foreign-key indexes should be an integral part of a database because they are straightforward to include and there is no good reason to defer them.

The database administrator (DBA) may define additional indexes for frequent queries and use product-specific tuning mechanisms.

19.4.4 Views

You may wish to define a view for each subclass to consolidate inherited data and make object access easier. A ***view*** is a table that an RDBMS dynamically computes. Figure 19.14 shows an example for the *CheckingAccount* subclass. You can freely read an object through a view, but RDBMSs only partially support writing through views. The restrictions vary across products.

```
CREATE VIEW view_checking_account AS
  SELECT chk_acct_ID, balance, credit_limit, protect_odrft
  FROM Account A, Checking_Account CA
  WHERE A.account_ID = CA.chk_acct_ID;
```

Figure 19.14 A sample RDBMS view. You can use a view to consolidate the object fragments that are stored for each generalization level.

19.4.5 Summary of Advanced Rules for RDBMS Implementation

Table 19.2 summarizes the advanced rules for implementing RDBMS structure.

Concept	Advanced implementation rule
Class	■ Define a check constraint for each enumerated attribute.
Association, Aggregation, Composition	■ Enforce foreign keys. Specify *on delete cascade* or *on delete no action*, depending on the model's meaning. ■ Define a check constraint for each enumerated attribute. ■ Define indexes for any buried foreign keys not covered by primary and candidate key constraints.
Generalization	■ Enforce foreign keys. Specify *on delete cascade* for each subclass table. ■ Define a check constraint for each generalization set name. ■ Consider defining a view to consolidate inherited data and to ease reading of objects.

Table 19.2 Summary of advanced rules for implementing relational databases. These rules are embedded in most database generation tools.

19.5 Implementing Structure for the ATM Example

Figure 19.15 puts together all the rules and shows the tables for Figure 19.2. Figure 19.16 shows SQL code that creates Oracle database structures. Each *sequence* statement creates a counter that is used to allocate an ID. For example, *seq_bank* is used to allocate *bank_ID* as each *Bank* object is created. We have organized the code logically, and it must be reordered before execution. First execute the create table and create sequence statements, then the create index statements, and finally the alter table statements. We have omitted the optional views on *CheckingAccount* and *SavingsAccount*.

Bank table

bankID	name (ck1)

Customer table

customerID	name	tempAmount

Account table

account ID	balance	credit Limit	bankID (ck1) (references Bank)	accountCode (ck1)	account Type	customerID (references Customer)

CardAuthorization table

card AuthorizationID	password	limit	bankID (ck1) (references Bank)	cardCode (ck1)	customerID (references Customer)

CashCard table

cashCardID	serialNumber	cardAuthorizationID (references CardAuthorization)

Account_CardAuthorization table

accountID (references Account)	**cardAuthorizationID** (references CardAuthorization)

SavingsAccount table

savingsAccountID (references Account)

CheckingAccount table

checkingAccountID (references Account)	protectFrom Overdraft

Address table

address ID	address	customerID (references Customer)

Figure 19.15 RDBMS tables for the abbreviated ATM model

```
CREATE TABLE Bank
( bank_ID    NUMBER(30)    CONSTRAINT nn_bank1 NOT NULL,
  bank_name  VARCHAR2(50)  CONSTRAINT nn_bank2 NOT NULL,
CONSTRAINT pk_bank PRIMARY KEY (bank_ID),
CONSTRAINT uq_bank1 UNIQUE (bank_name));
CREATE SEQUENCE seq_bank;

CREATE TABLE Customer
( customer_ID NUMBER(30)   CONSTRAINT nn_customer1 NOT NULL,
  cust_name   VARCHAR2(50) CONSTRAINT nn_customer2 NOT NULL,
  temp_amount NUMBER(12,2),
CONSTRAINT pk_customer PRIMARY KEY (customer_ID));
CREATE SEQUENCE seq_customer;

CREATE TABLE Card_Authorization
( card_auth_ID NUMBER(30)   CONSTRAINT nn_cardauth1 NOT NULL,
  password     VARCHAR2(50),
  limit        NUMBER(12,2),
  bank_ID      NUMBER(30)   CONSTRAINT nn_cardauth2 NOT NULL,
  card_code    VARCHAR2(50) CONSTRAINT nn_cardauth3 NOT NULL,
  customer_ID  NUMBER(30)   CONSTRAINT nn_cardauth4 NOT NULL,
CONSTRAINT pk_cardauth PRIMARY KEY (card_auth_ID),
CONSTRAINT uq_cardauth1 UNIQUE (bank_ID, card_code));
CREATE SEQUENCE seq_cardauth;
CREATE INDEX index_cardauth1 ON Card_Authorization
   (customer_ID);
ALTER TABLE Card_Authorization ADD CONSTRAINT fk_cardauth1
  FOREIGN KEY bank_ID
  REFERENCES Bank;
ALTER TABLE Card_Authorization ADD CONSTRAINT fk_cardauth2
  FOREIGN KEY customer_ID
  REFERENCES Customer;

CREATE TABLE Checking_Account
( chk_acct_ID   NUMBER(30)  CONSTRAINT nn_chkacct1 NOT NULL,
  protect_odrft VARCHAR2(1) CONSTRAINT nn_chkacct2 NOT NULL,
CONSTRAINT pk_chkacct PRIMARY KEY (chk_acct_ID));
ALTER TABLE Checking_Account ADD CONSTRAINT fk_chkacct1
  FOREIGN KEY chk_acct_ID
  REFERENCES Account ON DELETE CASCADE;
ALTER TABLE Checking_Account ADD CONSTRAINT enum_chkacct1
  CHECK (protect_odrft IN ('Y', 'N'));
```

Figure 19.16 SQL code for the abbreviated ATM model

```
CREATE TABLE Account
( account_ID   NUMBER(30)   CONSTRAINT nn_account1 NOT NULL,
  balance      NUMBER(12,2) CONSTRAINT nn_account2 NOT NULL,
  credit_limit NUMBER(12,2),
  bank_ID      NUMBER(30)   CONSTRAINT nn_account3 NOT NULL,
  account_code VARCHAR2(50) CONSTRAINT nn_account4 NOT NULL,
  account_type VARCHAR2(20) CONSTRAINT nn_account5 NOT NULL,
  customer_ID  NUMBER(30)   CONSTRAINT nn_account6 NOT NULL,
CONSTRAINT pk_account PRIMARY KEY (account_ID),
CONSTRAINT uq_account1 UNIQUE (bank_ID, account_code));

CREATE SEQUENCE seq_account;

CREATE INDEX index_account1 ON Account (customer_ID);

ALTER TABLE Account ADD CONSTRAINT fk_account1
  FOREIGN KEY bank_ID
  REFERENCES Bank;

ALTER TABLE Account ADD CONSTRAINT fk_account2
  FOREIGN KEY customer_ID
  REFERENCES Customer;

ALTER TABLE Account ADD CONSTRAINT enum_account1
  CHECK (account_type IN ('Checking_Account',
  'Savings_Account'));

CREATE TABLE Acct_CardAuth
( account_ID   NUMBER(30) CONSTRAINT nn_acctca1 NOT NULL,
  card_auth_ID NUMBER(30) CONSTRAINT nn_acctca2 NOT NULL,
CONSTRAINT pk_acctca PRIMARY KEY (account_ID, card_auth_ID));

CREATE INDEX index_acctca1 ON Acct_CardAuth (card_auth_ID);

ALTER TABLE Acct_CardAuth ADD CONSTRAINT fk_acctca1
  FOREIGN KEY account_ID
  REFERENCES Account;

ALTER TABLE Acct_CardAuth ADD CONSTRAINT fk_acctca2
  FOREIGN KEY card_auth_ID
  REFERENCES Card_Authorization;

CREATE TABLE Savings_Account
( sav_acct_ID NUMBER(30) CONSTRAINT nn_savacct1 NOT NULL,
CONSTRAINT pk_savacct PRIMARY KEY (sav_acct_ID));

ALTER TABLE Savings_Account ADD CONSTRAINT fk_savacct1
  FOREIGN KEY sav_acct_ID
  REFERENCES Account ON DELETE CASCADE;
```

Figure 19.16 (continued) **SQL code for the abbreviated ATM model**

```
CREATE TABLE Cash_Card
( cash_card_ID NUMBER(30)    CONSTRAINT nn_cashcard1 NOT NULL,
  serial_num   VARCHAR2(50) CONSTRAINT nn_cashcard2 NOT NULL,
  card_auth_ID NUMBER(30)    CONSTRAINT nn_cashcard3 NOT NULL,
CONSTRAINT pk_cashcard PRIMARY KEY (cash_card_ID),
CONSTRAINT uq_cashcard1 UNIQUE (serial_num));
CREATE SEQUENCE seq_cashcard;
CREATE INDEX index_cashcard1 ON Cash_Card (card_auth_ID);
ALTER TABLE Cash_Card ADD CONSTRAINT fk_cashcard1
  FOREIGN KEY card_auth_ID
  REFERENCES Card_Authorization;

CREATE TABLE Address
( address_ID  NUMBER(30)     CONSTRAINT nn_address1 NOT NULL,
  address     VARCHAR2(200) CONSTRAINT nn_address2 NOT NULL,
  customer_ID NUMBER(30)     CONSTRAINT nn_address3 NOT NULL,
CONSTRAINT pk_address PRIMARY KEY (address_ID));
CREATE SEQUENCE seq_address;
CREATE INDEX index_address1 ON Address (customer_ID);
ALTER TABLE Address ADD CONSTRAINT fk_address1
  FOREIGN KEY customer_ID
  REFERENCES Customer ON DELETE CASCADE;
```

Figure 19.16 (continued) **SQL code for the abbreviated ATM model**

19.6 Implementing Functionality

Most often your purpose in using a database will be to build an application. Structural concerns dominate a database, but the application's functionality—the user interface, complex logic, and other behavior—is also important. The use of UML models is an important first step toward combining database and programming capabilities—UML models provide a uniform way for thinking about both aspects. However, there are some additional functionality issues to consider.

- Coupling a programming language to a database. [19.6.1]
- Converting data. [19.6.2]
- Encapsulation vs. query optimization. [19.6.3]
- Use of SQL code. [19.6.4]

19.6.1 Coupling a Programming Language to a Database

Relational databases and conventional programming languages have divergent styles that make them difficult to couple. Relational databases are declarative; developers describe the

data they want instead of how to get it. In contrast, most programming languages are imperative and require that logic be reduced to a sequence of steps. Many techniques are available for combining databases and programming languages, and it is important that you consider all of your options.

- **Preprocessor and postprocessor**. Preprocessors and postprocessors can be helpful for batch applications. The basic idea is simple: Query the database and create an input file, run the application, and then analyze the output and store the results in the database.

  The downside is that database interaction via intermediate files can be awkward. The preprocessor must request all database information before executing the application, and output files with complex formats can be difficult to parse. This technique is useful for old software or certified software that cannot be altered.

- **Script files**. Sometimes all you need is a file of DBMS commands. For example, typing *@filename* into interactive SQL (SQL Plus) of Oracle causes the commands in *filename* to execute. Developers can use an operating system shell language to execute multiple script files and to control their execution.

  Script files are helpful for simple database interaction, such as creating database structures. They are also useful for prototyping.

- **Embedded DBMS commands**. Another technique is to intersperse SQL commands with application code.

  Many database books emphasize this technique. Unfortunately, such programs can be difficult to read and maintain. The essential problem is that the conceptual basis for an RDBMS is different than that for most programming languages. We discourage the use of this approach.

- **Custom application programming interface (API)**. A better alternative to embedded DBMS commands is to encapsulate database read and write requests within dedicated application methods that collectively provide a database interface. You can write these methods using a proprietary database-access language, such as Oracle's PL/SQL, or a standard language, such as ODBC or JDBC.

  A custom API isolates database interaction from the rest of the application. An API can help you partition the tasks of data management, application logic, and user interface consistent with the spirit of encapsulation.

- **Stored procedures**. A ***stored procedure*** is programming code that is stored in a database.

  Some RDBMSs, such as SQL Server, require stored procedures for maximum efficiency. Stored procedures also let applications share general-purpose functionality. However, you should try to avoid placing application-specific functionality in stored procedures—once the logic is placed in the database, any application can use it, compromising encapsulation. Stored procedures also vary widely across products.

- **Fourth-generation language (4GL)**. A ***fourth-generation language*** is a framework for database applications that provides screen layout, simple calculations, and reports.

  4GLs are widely available and can greatly reduce application development time. They are best for straightforward applications and prototyping. They are not suitable for applications with complex programming.

- **Generic layer**. A generic layer hides the DBMS and provides simple data access commands (such as *getRecordGivenKey* and *writeRecord*) [Blaha-98]. You can write application code in terms of the layer and largely ignore the underlying DBMS.

  A well-conceived generic layer can simplify application programming. However, it can also impede performance and restrict access to database functionality.

- **Metadata-driven system**. The application indirectly accesses data by first accessing the data's description (the metadata) and then formulating the query to access the data. For example, an RDBMS processes commands by accessing the system tables first and then the actual data.

  Metadata-driven applications can be quite complex. This technique is suitable for frameworks (see Chapter 14) and applications that learn.

Sometimes it is helpful to mix techniques. For example, a developer could use an API and implement some functionality with stored procedures. Table 19.3 summarizes the coupling techniques.

Data interaction technique	Recommendation
Preprocessor and postprocessor	Consider for old batch software or certified software that cannot be altered.
Script files	Consider for simple database interaction and prototyping.
Embedded DBMS commands	Use only when necessary. An API provides a better approach.
Custom API	Often a good choice.
Stored procedures	Good for general-purpose logic. Try to avoid placing application-specific logic in stored procedures.
4GL	Consider for straightforward applications and prototyping.
Generic layer	Consider when performance is not demanding and simple database functionality is needed.
Metadata driven	Consider for frameworks and other special situations.

Table 19.3 Data interaction techniques. It is important to consider all options for combining databases and programming languages.

19.6.2 Data Conversion

Legacy data processing is important for seeding new applications and exchanging data among applications. Many applications are poorly conceived, so it can be challenging and time consuming to rework their data. There are several key issues.

- **Cleansing data**. You must repair errors in source data. Errors arise from user mistakes, modeling flaws, database flaws, and application program errors. For example, an application program may have missed some addresses with illegal postal codes. A combination of fields might be intended to be unique, but the data may have errors if the database structure does not enforce uniqueness.
- **Handling missing data**. You must decide how to handle missing data. Can you find it elsewhere, do you want to estimate it, or can you use null values? You might want users to help resolve missing data.
- **Moving data**. It is common to migrate data from one application to another, either on a one-shot or a recurring basis. A standard language, such as XML, can be helpful for handling such data interchange.
- **Merging data**. Data sources may overlap. For example, one system may contain account information and another may contain address data. A new application may need both. For complex sources, it is best to model them first and then decide how to merge them.
- **Changing data structure**. Typically, source structures differ from target structures, so you must adjust the data. For example, one application may store a phone number in a single field; another may split country code, area code, and local phone number into separate fields. Corresponding fields may have different names, data types, and lengths. There may be different data encodings; for example, sex can be encoded as male or female, M or F, 1 or 2, and so on.

You should begin processing by loading the data into staging tables that mirror the original structure. For example, if the old application uses files and the new application uses a relational database, create one staging table for each file. Each column in a file maps to one column in a table with the same data type and length. Most RDBMSs have commands that readily perform this kind of loading.

The staging tables get the data into the database so that it can be operated upon with SQL commands. It is often better (less work, fewer errors, easier modification) to use SQL commands than to write custom programming code. Staging tables enable the full power of database queries to convert data from the old to the new format.

Often you can find alternative data sources. Customer data may be available from sales records, customer service records, and an external marketing firm, for example. To resolve redundancy, load the most accurate source first and then load the next most accurate and so on. Before loading each source, place it in a staging table so that SQL can eliminate already existing records. If you do not do this, you could load the same customer twice, for example. This approach is a simple way to resolve conflicts in data, and it biases the database toward the best sources.

19.6.3 Encapsulation vs. Query Optimization

Section 15.10.1 emphasized the importance of encapsulation (information hiding) and consequently the need to limit class-model traversals. Unfortunately, there is a conflict between the goals of encapsulation and the goals of RDBMS query optimization.

According to the principle of encapsulation, an object should access only directly related objects. Indirectly related objects should be accessed indirectly via the methods of intervening objects. Encapsulation increases the resilience of an application; local changes to an application model cause local changes to the application code.

On the other hand, RDBMS optimizers take a logical request and generate an efficient execution plan. If queries are broadly stated, the optimizer has greater freedom for devising an efficient plan. RDBMS performance is usually best if you join multiple tables together in a single SQL statement, rather than disperse logic across multiple SQL statements.

Thus encapsulation boosts resilience but limits optimization potential. In contrast, broadly stating queries enables optimization, but a small change to an application can affect many queries. For RDBMS applications, there is no simple resolution of this conflict. There are three different situations.

- **Complex programming**. You should encapsulate your code if the programming is intricate and performance degradation is not too severe.
- **Easy programming and good query performance**. You should broadly state queries if doing so improves RDBMS performance and the programming code and queries are relatively easy to write—and rewrite if the class model changes.
- **Easy programming and poor query performance**. Somewhat paradoxically, you can sometimes improve performance by fragmenting queries. Query optimizers are imperfect, and occasionally you will need to guide the optimizer manually.

19.6.4 Use of SQL Code

You can always write programming code for methods, but sometimes SQL provides a better alternative. A skilled developer can write SQL code faster than programming code. Furthermore, SQL code can execute faster, has fewer defects (bugs), and is easier to extend. The performance of SQL code benefits from reduced communication traffic (computation is confined to the server) and robust internal RDBMS algorithms.

For example, referring to the full ATM model in Chapter 17, we might want to prepare a monthly statement of transactions for each account. We could query the database and bring the various pieces of data into memory. Alternatively, Figure 19.17 shows a SQL command that provides the core data for a statement all at once. The names preceded with a colon are programming variables that are passed into the SQL command.

19.7 Object-Oriented Databases

An ***object-oriented database*** is a persistent store of objects that mix data and behavior. With an ordinary programming language, objects cease to exist when the program ends; with an object-oriented database, objects persist beyond the confines of program execution. An ***object-oriented DBMS (OO-DBMS)*** manages the data, programming code, and associated structures that constitute an object-oriented database. In contrast to RDBMSs, OO-DBMSs vary widely in their syntax and capabilities.

```
SELECT T.date_time, U.amount, U.kind
FROM Bank B, Account A, Update U, Remote_Transaction T
WHERE B.bank_ID = A.bank_ID AND
      A.account_ID = U.account_ID AND
      U.transaction_ID = T.transaction_ID AND
      B.bank_name = :aBankName AND
      A.account_code = :anAccountCode AND
      T.date_time >= :aStartDate AND
      T.date_time <= :anEndDate
ORDER BY T.date_time;
```

Figure 19.17 Offloading functionality to SQL code. Sometimes it is better to use SQL to implement a method than to write programming code.

OO-DBMSs are a relative newcomer to the database market. RDBMSs were commercialized in the 1970s, but OO-DBMSs were not introduced until the 1990s. Two major motivations led to the development of OO-DBMSs.

- **Programmer frustration with RDBMSs.** Many programmers don't understand RDBMSs and want something more familiar. RDBMSs are declarative (queries describe properties that requested data must satisfy), while most languages are imperative (stated as a sequence of steps). Furthermore, RDBMSs awkwardly combine with most languages, and programmers prefer a DBMS with a more seamless interface.

  This is a poor reason for choosing an OO-DBMS. The reality is that RDBMSs dominate the marketplace now and will for the foreseeable future. Programmers should not be using an OO-DBMS out of frustration; they should learn to deal with RDBMSs. RDBMS products are more mature and have proven features for reliability, scalability, and administration.

- **Need for special features.** RDBMSs lack the power that some advanced applications need. OO-DBMSs offer advanced features, like rich data types and quick access to low-level primitives.

  This is a good reason for considering an OO-DBMS. If you have an advanced application that is critical to your business, an OO-DBMS may ease development. Engineering applications, multimedia systems, and artificial intelligence software can sometimes benefit from the use of an OO-DBMS.

You should be selective about deciding to use an OO-DBMS. OO-DBMSs are not popular in the marketplace. OO-DBMS sales are only about 2% of RDBMS sales and have hit a plateau [Leavitt-00]. Consequently, you should use OO-DBMSs only for compelling situations.

19.8 Practical Tips

Here are tips for using a relational database to implement an OO design.

- **Normal forms**. Normal forms apply regardless of the development approach. However, it is unnecessary to check them if you build a sound OO model. (Section 19.1.3)
- **Classes**. Map each class to a table and each attribute to a column. (Section 19.3.1)
- **Associations**. For simple one-to-one and one-to-many associations, use a buried foreign key. For all other associations, use a distinct table. (Section 19.3.2)
- **Generalizations**. For single inheritance, map the superclass and subclasses each to a table. (Section 19.3.3)
- **Identity**. We strongly advise that you use object identity. Doing so has several advantages and little disadvantage. (Section 19.3.4)
- **Foreign keys**. Define constraints to enforce all foreign keys. For each subclass, specify *on delete cascade* for the foreign key to the superclass. For some association foreign keys, you may also want to specify *on delete cascade*. (Section 19.4.1)
- **Enumerations**. Use SQL check constraints to enforce them. (Section 19.4.2)
- **Indexes**. Create an index for every foreign key that is not covered by a primary key or candidate key constraint. You may want to define additional indexes for frequent queries and use product-specific tuning mechanisms. (Section 19.4.3)
- **Views**. You may want to define views to reconstitute objects that are fragmented across generalization tables. Such views are convenient for reading, but RDBMSs only partially support writing through views. (Section 19.4.4)
- **Coupling to a programming language**. Be deliberate about coupling a programming language to a database. Consider all your options. (Section 19.6.1)
- **Data conversion**. It is often helpful to load data into temporary staging tables. Then you can write SQL code to do much of the data processing. SQL code is easier and faster to write than programming code. (Section 19.6.2)
- **Encapsulation vs. query optimization**. Be aware of the intrinsic conflict between these two goals and make your best resolution on a case-by-case basis. (Section 19.6.3)
- **Object-oriented databases**. Consider them only when application needs are compelling. (Section 19.7)

19.9 Chapter Summary

A database management system (DBMS) is software that provides general-purpose functionality for storing, retrieving, and controlling access to data. A DBMS protects data from accidental loss and makes it available for sharing. Several paradigms are available, but the development of new applications is dominated by relational DBMSs (RDBMSs). OO-DBMSs are also available but pragmatic concerns limit their use.

OO models provide an excellent basis for thinking about databases. Developers can think about a problem abstractly and defer the details of design and implementation. With a proper implementation, sound OO models lead to extensible, efficient, and understandable databases. Table 19.1 and Table 19.2 summarize the implementation rules for RDBMS structure.

You should be deliberate in coupling a programming language to a database. Furthermore, you should look for opportunities to substitute SQL code for programming effort.

candidate key	implementing generalizations
coupling a language to a database	index
data conversion	normal form
database	null
database management system (DBMS)	object-oriented DBMS (OO-DBMS)
foreign key	primary key
identity	relational DBMS (RDBMS)
implementing associations	SQL
implementing classes	view

Figure 19.18 Key concepts for Chapter 19

Bibliographic Notes

Many good books explain DBMS and RDBMS principles. [Elmasri-00] is a premier textbook that explains database concepts. [Chaudhri-98] has thoughtful examples of applications that use OO databases.

[Blaha-98] elaborates the material in this chapter and provides more details for files, RDBMSs, and OO-DBMSs (specifically ObjectStore). Our approach to databases is consistent with that of other authors, such as [Muller-99].

[Chang-03] presents middleware for combining databases with GUIs via intermediate text files.

References

[Blaha-98] Michael Blaha and William Premerlani. *Object-Oriented Modeling and Design for Database Applications*. Upper Saddle River, NJ: Prentice Hall, 1998.

[Chang-03] Peter H. Chang. A platform independent middleware architecture for accessing databases on a database server on the Web. *IEEE Conference on Electro/Information Technology*. Indianapolis, 2003.

[Chaudhri-98] Akmal B. Chaudhri and Mary Loomis. *Object Databases in Practice*. Upper Saddle River, NJ: Prentice Hall PTR, 1998.

[Elmasri-00] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems, Third Edition*. Redwood City, CA: Benjamin/Cummings, 2000.

[Leavitt-00] Neal Leavitt. Whatever happened to object-oriented databases? *IEEE Computer*, August 2000, 16–19.

[Melton-93] Jim Melton and Alan R. Simon. *Understanding the New SQL: A Complete Guide*. San Francisco: Morgan Kaufmann, 1993.

[Muller-99] Robert J. Muller. *Database Design for Smarties: Using UML for Data Modeling*. San Francisco: Morgan Kaufmann, 1999.

Exercises

19.1 (8) Figure E19.1 shows four different class models for directed graphs. A directed graph consists of a set of edges and a set of nodes. Each edge connects two nodes and has an arrow indicating direction of flow. Any number of edges may connect to the same node. More than one edge may connect a pair of nodes. An edge may connect a node to itself.

In Figure E19.1a a graph is a many-to-many association between nodes with directionality indicated by *from* and *to* ends. In Figure E19.1b a graph is a many-to-many association between edges. The qualifiers, *end1* and *end2*, are enumerated types with possible values of *to* and *from* indicating which ends of the edges connect. Figure E19.1c treats both nodes and edges as objects. Two associations, *to* and *from*, store connections, one for each end of an edge. Figure E19.1d represents each connection as a qualified association. Each end of an edge connects to exactly one node, and *end* is an enumerated type.

Which diagram most accurately models a graph? Explain the relative merits of each diagram. What happens if more than one edge connects a given pair of nodes? Can an edge connect a node with itself? What happens if only one edge connects to a node?

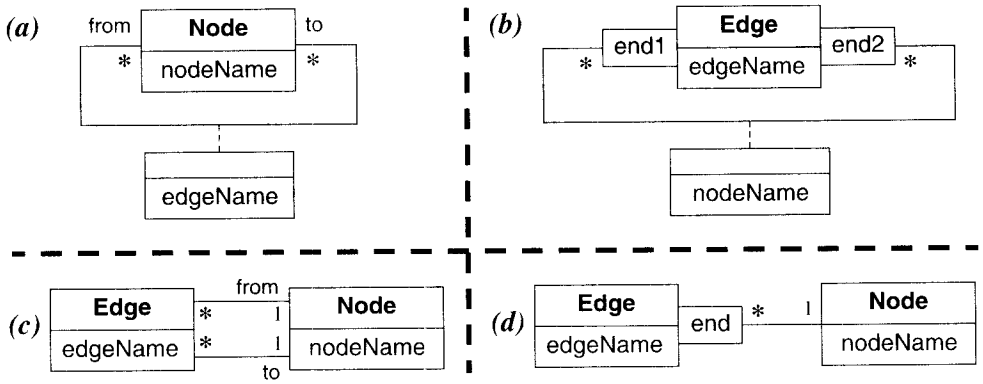


Figure E19.1 Alternative class models for a directed graph

19.2 (6) Prepare tables for each model from the previous exercise. Be sure to show primary, candidate, and foreign keys. Use object identity.

19.3 (4) Write SQL code to create an empty database for the tables for Figure E19.1c and Figure E19.1d from the previous exercise. Use your judgment to supply any missing information.

19.4 (3) Populate the database tables created by the SQL commands of Exercise 19.3 for the directed graph in Figure E19.2.

19.5 For the class model in Figure E19.1d, prepare SQL queries for the following. For part (d) you will need to augment SQL with pseudocode.

a. (3) Given the name of an edge, determine the two nodes that it connects.

b. (3) Given the name of a node, determine all edges connected to or from it.

c. (5) Given a pair of nodes, determine the edges, if any, that directly connect them in either direction.

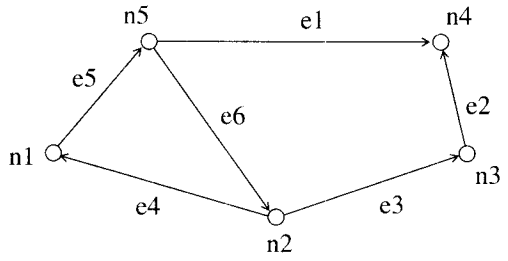


Figure E19.2 Sample directed graph

d. (8) Given a node, determine the nodes that can be visited directly or indirectly from the given node by traversing one or more edges (transitive closure). Each edge must be traversed from its *from* end to its *to* end.

19.6 (6) Prepare tables for Figure E19.3. An expression is a binary tree of terms that is formed from constants, variables, and arithmetic operators. Unary minus is not allowed.

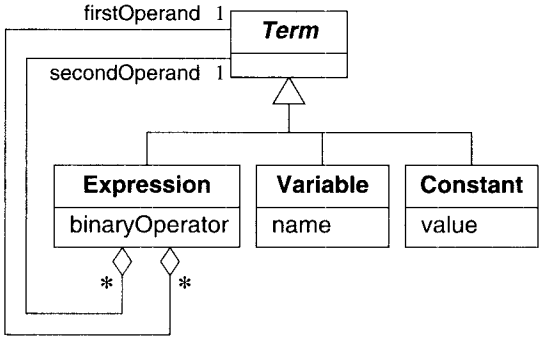


Figure E19.3 Class model for expressions

19.7 (4) Write SQL code to create an empty database for the tables from the previous exercise.

19.8 (5) Show populated database tables created by the previous exercise for the expression $(X + Y/2)/(X/3 - Y)$. Consider the parentheses in establishing the precedence of operators; otherwise, ignore them in populating the database tables.

19.9 (7) Prepare tables for Figure E19.4. A document consists of numbered pages. Each page contains many drawing objects—ellipses, rectangles, polylines, textlines, and groups of objects. Ellipses and rectangles are embedded within a bounding box. A polyline is a series of line segments defined by vertex points. Textlines originate at a point and have a font. Treat all associations and aggregations as unordered. For this exercise disregard the ordering of points in a polygon.

19.10 (7) Revise your tables from the previous exercise to treat the *Polyline_Point* association as ordered. That is, given a polyline, the database must be able to retrieve the points in the correct

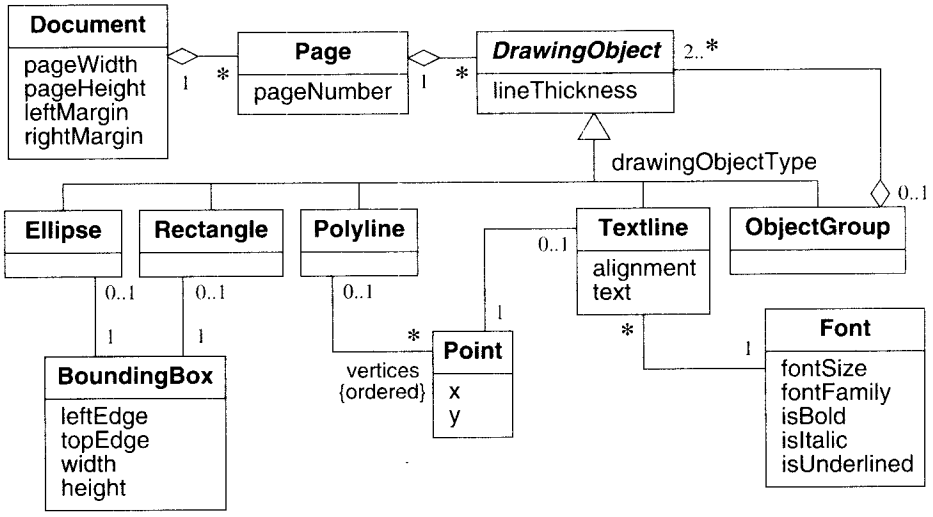


Figure E19.4 Class model for a desktop publishing system

order. [Instructor's note: You may want to give the students the answer to the previous exercise.]

19.11 (6) Modify your answer to Exercise 19.9 to reflect the revised class model in Figure E19.5. Discuss the merits of the revision. Disregard the ordering of *Points* in this exercise.

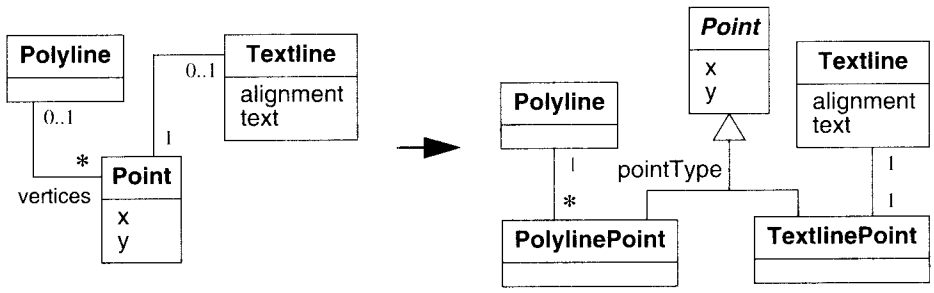


Figure E19.5 Generalization of point to eliminate zero-or-one multiplicity

19.12 (5) Write SQL code to create an empty database for the tables from Exercise 19.9.

19.13 (5) Convert the SQL commands in Figure E19.6 into a class model. The tables store the straight-line distances between pairs of cities.

19.14 (4) Using the tables from Exercise 19.13, write an SQL query that will determine the distance between two cities, given the names of the two cities.

```
CREATE TABLE City
( city_ID    NUMBER(30)     CONSTRAINT nn_city1 NOT NULL,
  city_name  VARCHAR2(255)  CONSTRAINT nn_city2 NOT NULL,
CONSTRAINT pk_city PRIMARY KEY (city_ID),
CONSTRAINT uq_city1 UNIQUE (city_name));
CREATE SEQUENCE seq_city;

CREATE TABLE Route
( route_ID  NUMBER(30)      CONSTRAINT nn_route1 NOT NULL,
  distance  NUMBER(20,10))  CONSTRAINT nn_route2 NOT NULL,
CONSTRAINT pk_route PRIMARY KEY (route_ID));
CREATE SEQUENCE seq_route;

CREATE TABLE City_Distance
( city_ID   NUMBER(30)  CONSTRAINT nn_dist1 NOT NULL,
  route_ID  NUMBER(30)  CONSTRAINT nn_dist2 NOT NULL,
CONSTRAINT pk_dist PRIMARY KEY (city_ID, route_ID));
CREATE INDEX index_dist1 ON City_Distance (route_ID);
ALTER TABLE City_Distance ADD CONSTRAINT fk_dist1
  FOREIGN KEY city_ID
  REFERENCES City;
ALTER TABLE City_Distance ADD CONSTRAINT fk_dist2
  FOREIGN KEY route_ID
  REFERENCES Route;
```

Figure E19.6 SQL commands for creating tables to store distances between cities

19.15 (5) Convert the SQL commands in Figure E19.7 into a class model. The tables store the straight-line distances between pairs of cities.

19.16 (6) Using the tables in Exercise 19.15, write an SQL query that will determine the distance between two cities, given their names. Assume that the distance between a given pair of cities is stored exactly once in the *CityDistance* table. (The application must enforce a constraint such as *city1ID* < *city2ID*, so that the distance is entered only once.)

19.17 (6) Discuss the relative merits of the two approaches in the previous four exercises for storing distance information.

19.18 (5) Discuss the similarities and differences between the database tables used to store edge and node information in Exercises 19.1–19.5 and the tables used to store distance information between cities in Exercises 19.13–19.17. How does fact that there is exactly one straight-line distance between a pair of cities simplify the problem? Is the problem of storing distances between cities more nearly like a directed graph or an undirected graph? Why?

```
CREATE TABLE City
( city_ID   NUMBER(30)     CONSTRAINT nn_city1 NOT NULL,
  city_name VARCHAR2(255) CONSTRAINT nn_city2 NOT NULL,
CONSTRAINT pk_city PRIMARY KEY (city_ID),
CONSTRAINT uq_city1 UNIQUE (city_name));
CREATE SEQUENCE seq_city;

CREATE TABLE City_Distance
( city1_ID  NUMBER(30)     CONSTRAINT nn_dist1 NOT NULL,
  city2_ID  NUMBER(30)     CONSTRAINT nn_dist2 NOT NULL,
  distance  NUMBER(20,10) CONSTRAINT nn_dist3 NOT NULL,
CONSTRAINT pk_dist PRIMARY KEY (city1_ID, city2_ID));
CREATE INDEX index_dist1 ON City_Distance (city2_ID);
ALTER TABLE City_Distance ADD CONSTRAINT fk_dist1
  FOREIGN KEY city1_ID
  REFERENCES City;
ALTER TABLE City_Distance ADD CONSTRAINT fk_dist2
  FOREIGN KEY city2_ID
  REFERENCES City;
```

Figure E19.7 SQL commands for creating tables to store distances between cities

19.19 (7) Prepare tables and SQL commands to create an empty relational database for the model in Figure E19.8.

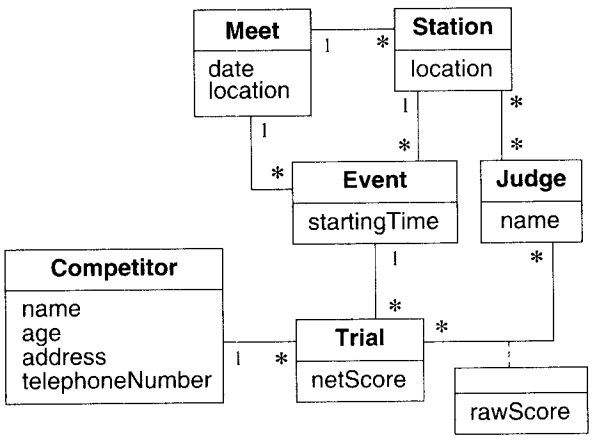


Figure E19.8 Partial class model for a scoring system

19.20 (6) Build a class model of forks and philosophers for the dining philosophers problem. (See Exercise 3.28.) Prepare tables and SQL code to create an empty database for your model. Show database contents for the situation in which each philosopher has exactly one fork.

19.21 (7) Prepare tables and SQL commands to create an empty relational database for the model in Figure 3.27.

19.22 (8) Write an SQL query for each of the OCL expressions in Section 3.5.3.

20

Programming Style

As any chess player, cook, or skier can attest, there is a great difference between knowing something and doing it well. Writing programs is no different. It is not enough to know the basic constructs and to be able to assemble them. The experienced programmer follows principles to make readable programs that live beyond the immediate need. These principles include programming idioms, rules of thumb, tricks of the trade, and cautionary advice. Good style is important in all programming, but it is even more important in OO programming, because much of the benefit of the OO approach is predicated on producing reusable, extensible, and understandable programs.

20.1 Object-Oriented Style

Good programs do more than simply satisfy their functional requirements. Programs that follow proper coding guidelines are more likely to be correct, reusable, extensible, and quickly debugged. Most style guidelines that are intended for conventional programs also apply to OO programs. In addition, facilities such as inheritance are peculiar to OO languages and require new guidelines. We present OO style guidelines under the following categories, although many guidelines contribute to more than one category.

- Reusability [20.2]
- Extensibility [20.3]
- Robustness [20.4]
- Programming-in-the-Large [20.5]

20.2 Reusability

Reusable software reduces design, coding, and testing cost by amortizing effort over several applications. Reducing the amount of code also simplifies understanding, which increases

the likelihood that the code is correct. Reuse is possible in conventional languages, but OO languages greatly enhance the possibility of code reuse.

20.2.1 Kinds of Reusability

There are two kinds of reuse: sharing of newly written code within an application and reuse of previously written code on new applications. Similar guidelines apply to both kinds of reuse. Sharing of code within an application is a matter of discovering redundant code sequences and using programming-language facilities to consolidate them (refactoring). This kind of code sharing almost always produces smaller programs and faster debugging.

Planning for future reuse takes more foresight and represents an investment. It is unlikely that a class in isolation will be used for multiple applications. Programmers are more likely to reuse carefully thought out subsystems, such as abstract data types, graphics packages, and numerical analysis libraries. Patterns and frameworks can be helpful in this regard (see Chapter 14).

20.2.2 Style Rules for Reusability

There are a number of rules that you can follow to promote reusability within your application and across applications.

- **Keep methods coherent**. A method is coherent if it performs a single function or a group of closely related functions. If it does two or more unrelated things, break it apart into smaller methods.
- **Keep methods small**. If a method is large, break it into smaller methods. A method that exceeds one or two pages is probably too large. By breaking a method into smaller parts, you may be able to reuse some parts even when the entire method is not reusable.
- **Keep methods consistent**. Similar methods should use the same names, argument order, data types, return value, and error conditions. Maintain parallel structure when possible. The methods for an operation should have consistent signatures and semantics.

  The Unix operating system offers many examples of inconsistent functions. For example, in the C library, there are two inconsistent functions to output strings, *puts* and *fputs*. The *puts* function writes a string to the standard output, followed by a newline character; *fputs* writes a string to a specified file, without a newline character. Avoid such inconsistency.
- **Separate policy and implementation**. Policy methods make decisions, shuffle arguments, and gather global context. Policy methods switch control among implementation methods. Policy methods should check for status and errors; they should not directly perform calculations or implement complex algorithms. Policy methods are often application dependent, but they are simple to write and easy to understand.

  Implementation methods perform specific detailed logic, without deciding whether or why to do them. If implementation methods can encounter errors, they should only return status, not take action. Implementation methods perform specific computations

on fully specified arguments and often contain complicated algorithms. Implementation methods do not access global context, make decisions, contain defaults, or switch flow of control. Because implementation methods are self-contained algorithms, they are likely to be meaningful and reusable in other contexts.

Do not combine policy and implementation in a single method. Isolate the core of the algorithm into a distinct, fully specified implementation method. This requires abstracting out the particular parameters of the policy method as arguments in a call to the implementation method.

For example, a method to scale a window by a factor of 2 is a policy method. It should set the target scale factor for the window and call on an implementation method that scales the window by an arbitrary scale factor. If you decide later to change the default scale factor to another value, such as 1.5, you just have to modify the parameter in the policy method, without changing the implementation method that actually does the work.

- **Provide uniform coverage**. If input conditions can occur in various combinations, write methods for all combinations, not just the ones that you currently need. For example, if you write a method to get the last element of a list, also write one to get the first element. By providing uniform coverage you not only boost reusability, but you also rationalize the scope of related methods.
- **Broaden the method as much as possible**. Try to generalize argument types, preconditions and constraints, assumptions about how the method works, and the context in which the method operates. Take meaningful actions on empty values, extreme values, and out-of-bounds values. Often a method can be made more general with a slight increase in code.
- **Avoid global information**. Minimize external references. Referring to a global object imposes required context on the use of a method. Often the information can be passed in as an argument. Otherwise store global information as part of the target object so that other methods can access it uniformly.
- **Avoid methods with state**. Methods that drastically change behavior depending on previous method history are hard to reuse. Try to replace them with stateless methods. For example, a text-processing application requires insert and replace operations. One approach is to set a flag to *insert* or *replace*, then use a *write* operation to insert or replace text depending on the current flag. A stateless approach uses two operations, *insert* and *replace*, that do the same operations without a state setting. The danger of method states is that an object left in a state in one part of an application can affect a method applied later in the application.

20.2.3 Using Inheritance

The preceding guidelines improve the chance of sharing code. Sometimes, however, methods on different classes are similar but not similar enough to represent with a single inherited method. There are several techniques of breaking up methods to inherit some code.

- **Factor out commonality**. The simplest approach is to factor out the common code into a single method that is called by each method. The common method can be assigned to an ancestor class. This is effectively a subroutine call, as Figure 20.1 shows.

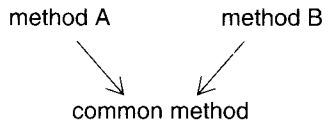


Figure 20.1 Code reuse via factoring out commonality. Place the common code into a method that is called by the original methods.

- **Factor out differences**. In some cases the best way to increase code reuse between similar classes is to factor out the differences between the methods of different classes, leaving the remainder of the code as a shared method. This approach is effective when the differences between methods are small and the similarities are great. As Figure 20.2 shows, the common portion of two methods is made into a new method. The new method calls an operation that is implemented by different methods containing the code differences for each subclass. Sometimes an abstract class must be added to hold the top-level method. This approach makes it easier to add new subclasses, because only the difference code need be written.

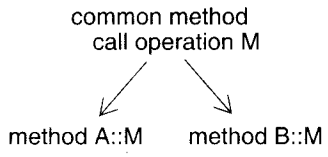


Figure 20.2 Code reuse via factoring out differences. Place the difference code in polymorphic methods of a common operation.

A package for plotting numerical data illustrates factoring. *DataGraph* is an abstract class that organizes common data and operations for its subclasses. One of *DataGraph's* methods is *draw*, consisting of the following steps: draw border, scale data, draw axes, plot data, draw title, and draw legend.

Subclasses of *DataGraph*, such as *LineGraph*, *BarGraph*, and *ScatterGraph*, draw borders, titles, and legends the same way but differ in the way they scale data, draw axes, and plot data. Each subclass inherits the methods *drawBorder*, *drawTitle*, and *drawLegend* from abstract class *DataGraph*, but each subclass defines its own methods for *scaleData*, *drawAxes*, and *plotData*. The method *draw* need be defined only once, on class *DataGraph*, and is inherited by each subclass. Each time the draw method is invoked, it applies *drawBorder*, *drawTitle*, and *drawLegend* inherited from the superclass and *scaleData*, *drawAxis*, and *plotData* supplied by the subclass itself. To add a new subclass, only the three specialized methods need be written.

- **Delegate**. Sometimes it appears that use of inheritance would increase code reuse within a program, when a true superclass/subclass relationship does not exist. Do not give in to the temptation to use this *implementation inheritance*; use delegation instead. Inheritance should be used only when the generalization relationship is semantically valid. Inheritance means that each instance of a subclass truly is an instance of the superclass; thus all operations and attributes of the superclass must uniformly apply to the subclasses. Improper use of inheritance leads to programs that are hard to maintain and extend. OO languages are permissive in their use of inheritance and will not enforce the good programming practice that we recommend.

  Delegation provides a proper mechanism to achieve the desired code reuse. The method is caught in the desired class and forwarded to another class for actual execution. Since each method must be explicitly forwarded, unexpected side effects are less likely to occur. The names of methods in the catching class may differ from those in the supplier class. Each class should choose names that are most appropriate for its purposes.

- **Encapsulate external code**. Often you will want to reuse code that may have been developed for an application with different interfacing conventions. Rather than inserting a direct call to the external code, it is safer to encapsulate its behavior within a method or a class. This way, the external routine or package can be changed or replaced, and you will have to change your code in only one place.

  For example, you may have a numerical analysis application but, knowing that reliable matrix-inversion software already exists, you do not want to reimplement the algorithm. You could write a matrix class to encapsulate the functionality provided by the external subroutine package. The matrix class would have, for example, an inverse method that takes the tolerance-for-singularity as an argument and returns a new matrix that is the inverse of the method's target.

20.3 Extensibility

Most software is extended in ways that its original developers never expect. The reusability guidelines enhance extensibility, as do the additional guidelines listed below.

- **Encapsulate classes**. A class is encapsulated if its internal structure is hidden from other classes. Only methods on the class should access its implementation. Many compilers are smart enough to optimize methods into direct access to the implementation, but the programmer should not. Respect the information in other classes by never reaching inside the class for data.

- **Specify visibility for methods**. Public methods are visible outside a class and have published interfaces. Once a public method is used by other classes, it is costly to change its interface, so carefully define and limit the number of public methods. In contrast, private methods are internal to a class—they can be deleted or changed with impact limited to other methods on the class. Protected methods have intermediate visibility—see Section 18.1.

Careful use of visibility (see Section 4.1.4) makes your classes easier to understand and increases code resilience. Private and protected methods suppress unnecessary details from the user of a class, avoiding confusion. Unlike a public method, a private method cannot be applied out of context, so it can rely on preconditions or state information of the class.

- **Hide data structures.** Do not export data structures from a method. Internal data structures are specific to a method's algorithm. If you export them, you limit flexibility to change the algorithm later.
- **Avoid traversing multiple links or methods.** A method should have limited knowledge of a class model. A method must be able to traverse links to obtain its neighbors and must be able to call methods on them, but it should not traverse a second link from the neighbor to a third class because the second link is not directly visible to it. Instead, call a method on the neighbor object to traverse nonconnected objects; if the association network changes, the method can be rewritten without changing the call.

  Similarly, avoid applying a second method to the result of a method call unless the class of the result is already known as an attribute, argument, or neighbor, or the result class is from a lower-level library. Instead, write a new method on the original target class to perform the combined method itself. The principles in this bullet were proposed in [Lieberherr-89] as the "Law of Demeter."
- **Avoid case statements on object type.** Use polymorphism instead. Case statements can be used to test internal attributes of an object but should not be used to select behavior based on object type. The dispatching of operations based on object type is the whole point of polymorphism, so don't circumvent it.
- **Design for portability.** Limit system dependencies to a few basic methods. You can then more easily port software to other hardware platforms and software environments. For example, if you are using a database, you might isolate database access. This would make it easier to switch database vendors and possibly enable switching paradigm (such as from relational to OO database).

20.4 Robustness

You should strive for efficient methods but not at the expense of robustness. A method is robust if it does not fail even if it receives improper parameters. In particular, you should never sacrifice robustness against user errors.

- **Protect against errors.** Software should protect itself against incorrect user input and never let it cause a crash. Methods must validate input that could cause trouble.

  There are two kinds of errors. Analysis uncovers errors that exist in the problem domain and determines an appropriate response. For example, an ATM should handle errors for the card scanner and communications lines. On the other hand, low-level system errors concern programming aspects. These low-level errors include memory allocation

errors, file input/output errors, and hardware faults. Your program should check for system errors and at least try to die gracefully if nothing else is possible.

Try to guard against programming bugs as well, and give good diagnostic information. During development, it is often worthwhile to insert internal assertions into the code to uncover bugs, even though the checks will be removed for efficiency in the production version. A strongly typed OO language provides greater protection against type mismatches, but you can insert assertions in any language. In particular, you should check array bounds.

- **Optimize after the program runs**. Don't optimize a program until you get it working. Many programmers spend too much time improving code that seldom executes. Measure the performance within the program first; you may be surprised to find that most parts consume little time. Study your application to learn what measures are important, such as worst-case times and method frequencies. If a method may be implemented in more than one way, assess the alternatives with regard to memory, speed, and implementation simplicity. In general, avoid optimizing more of the program than you have to, since optimization compromises extensibility, reusability, and understandability. If methods are properly encapsulated, you can replace them with optimized versions without affecting the rest of the program.
- **Always construct objects in a valid state** [Vermeulen-00]. Otherwise, you create a situation where someone might use the constructor and not call the right subsequent method. As a matter of good software practice, your code should always be logically sound.
- **Validate arguments**. Public methods, those available to users of the class, must rigorously check their arguments, because users may violate restrictions. Private and protected methods can improve efficiency by assuming that their arguments are valid, since the implementor can rely on the public methods that call them for error checking.

  Don't write or use methods whose arguments can't be validated. For example, the infamous *scanf* function in Unix reads a line of input into an internal buffer without checking the size of the buffer. This loophole has been exploited to write virus programs that force a buffer overflow in system software that did not validate its arguments.
- **Avoid predefined limits**. When possible use dynamic memory allocation to create data structures without predefined limits. It is difficult to predict the maximum capacity expected of data structures in an application, so don't set any limits. The day of fixed limits on symbol table entries, user names, file names, compiler entries, and other things should be long over. Most OO languages have excellent dynamic memory allocation facilities.
- **Instrument the program for debugging and performance monitoring**. Just as a hardware circuit designer instruments an IC board with test points, you should instrument your code for testing, debugging, statistics, and performance. The level of debugging that you must build into your code depends on your language's programming environment. You can add debug statements to methods that conditionally execute depending on the debug level. The debug statements print a message on entry or exit and selective input and output values.

You can better understand the behavior of classes by adding code to gather statistics. Some operating systems, such as Unix, offer tools to create execution profiles of an application. Typically, these tools report the number of times each method was called and the amount of processor time spent in each method. If your system lacks comparable tools, you can instrument your code for gathering statistics much like for debugging.

20.5 Programming-in-the-Large

Programming-in-the-large refers to writing large, complex programs with teams of programmers. Human communication becomes paramount on such projects and requires proper software engineering practices. You should observe the following guidelines.

- **Do not program prematurely**. Think long and carefully before you commit to code. Ultimately, of course you must have code to deliver your application. But code is tedious to write and difficult to change. In contrast, models are much more malleable, because they are high level and suppress details. It is much better to work out your ideas with models, gain a full understanding, and only then write the code.
- **Make methods readable**. Meaningful variable names increase readability. Typing a few extra characters costs less than the misunderstanding that can come later when another programmer tries to decipher your variable names. Avoid confusing abbreviations, and use temporary variables instead of deeply nested expressions. Do not use the same temporary variable for two different purposes within a method, even if their usage does not overlap; most compilers will optimize this anyway. Minor improvements to efficiency are not worth compromising readability.
- **Keep methods understandable**. A method is understandable if someone other than the creator can understand the code (as well as the creator after a time lapse). Keeping methods small and coherent helps to accomplish this.
- **Use the same names as in the class model**. The names used within a program should match those in the class model. A program may need to introduce additional names for implementation reasons, but you should preserve the names for concepts that carry forward. This practice improves traceability, documentation, and understandability. It is reasonable to adopt conventions such as uniform prefixes for consistency across applications, thereby avoiding name clashes.
- **Choose names carefully**. Make sure that your names accurately describe the operations, classes, and attributes that they label. Follow a uniform style in devising names. For example, you may name operations as *"verbObject,"* such as *addElement* or *drawHighlight*. Be sure to define a vocabulary of verbs that are often used. For example, don't use both *new* and *create* unless they have different meanings. Many OO languages automatically build method names from the class and operation names.

  Similarly, do not use the same method name for semantically different purposes. As Figure 20.3 shows, all methods with the same name should have the same signature (number and types of arguments) and meaning.

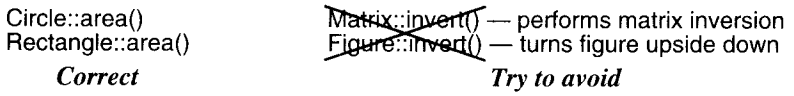


Figure 20.3 Method names. All methods with the same name should have the same signature and meaning.

- **Use programming guidelines.** Project teams should use programming guidelines available in their organizations or external guidelines, such as [Vermeulen-00]. Guidelines address issues such as the form of variable names, indentation style for control structures, method documentation headers, and in-line documentation.
- **Use packages.** Group closely related classes into a package. (See Section 4.11.)
- **Document classes and methods.** The documentation of a method describes its purpose, function, context, inputs, and outputs as well as any assumptions and preconditions about the state of the object. You should describe not only the details of an algorithm, but also why it was chosen. Internal comments within the method should describe major steps.
- **Publish the specification.** The specification is a contract between the producer and the consumer of a class. Once a specification is written, the producer cannot break the contract, for doing so would affect the consumer. The specification contains only declarations, and the user should be able to use the class just by looking at the declarations. Online descriptions of a class and its features help promote the correct use of the class. Figure 20.4, Figure 20.5, and Figure 20.6 show sample specifications.

Class name: Circle
Version: 1.0
Description: Ellipse whose major and minor axes are equal
Super classes: Ellipse
Features:
Private attributes:
center: Point — location of its center
radius: Real — its radius
Public methods:
draw (Window) — draws a circle in the window
intersectLine (Line): Set of Points — finds the intersection of a line and a circle, returns set of 0–2 points
area (): Real — calculates area of circle
perimeter (): Real — calculates circumference of circle
Private methods: none

Figure 20.4 Class specification

Operation: intersectLine (line: Line) : Set of Points

Origin Class: GeometricFigure

Description: Returns a set of intersection points between the geometric object and the line. The set may contain 0, 1, or more points. Each tangent point appears only once. If the line is collinear with a line segment in the figure, only the two end points of the segment are included.

Status: Abstract operation in the origin class, must be overridden.

Inputs:

self: GeometricFigure — figure to be intersected with line
line:Line — line to be intersected with circle

Returns:

A set of intersection points. Set may contain 0 or more points.

Side Effects: none

Errors:

If the figures do not intersect, returns an empty set.
If the line is collinear with a line segment in the figure, the set includes only the end points of the segment.
If the figure is an area, then its boundary is used

Figure 20.5 Operation specification

Method: Circle::intersectLine (line: Line) : Set of Points

Description: Given a circle and a line, finds the intersection, returns a set of 0–2 intersection points. If the line is tangent to the circle, the set contains a single point.

Inputs:

self:Circle — circle to be intersected with line
line:Line — line to be intersected with circle

Returns:

A set of intersection points. Set may contain 0, 1, or 2 points.

Side Effects: none

Errors:

If the figures do not intersect, returns an empty set.
If the line is tangent to the circle, returns the tangent point.
If the circle's radius is 0, returns a single point if the point is on the line.

Figure 20.6 Method specification

- **Avoid duplicated code**. [Baker-95] cites two applications with one million lines of code. (One was X-Windows.) She found that at least 12% of the code was gross duplication that could be easily eliminated. Such duplication comes from programmers copying and editing code when making bug fixes and other reasons of expedience. The side effect is that the code swells in size, making it more difficult to maintain and understand.

20.6 Chapter Summary

Good style is important to maximize the benefits of OO programming; most benefits come from greatly reduced maintenance and enhancement costs and the reuse of the new code for future applications. OO programming style guidelines include conventional programming style guidelines as well as principles uniquely applicable to OO concepts such as inheritance.

A major goal of OO development is maximizing reusability of classes and methods. Reuse within a program is a matter of looking for similarities and consolidating them. Planning for reuse by future applications takes more time and effort up front. Reusability is enhanced by keeping methods small, coherent, and local. Separation of policy and implementation is important. One way to use inheritance is by factoring a generic method into submethods, some inherited from the origin class and some provided by each subclass. Delegation should be used when methods must be shared but classes are not in a true generalization relationship.

Most software is eventually extended. Extensibility is enhanced by encapsulation, minimizing dependencies, using methods to access attributes of other classes, and distinguishing the visibility of methods.

Do not sacrifice robustness for efficiency. Because objects contain references to their own classes, they are less vulnerable to mismatched typing than conventional programming variables and can be checked dynamically to see that they match the assumptions within a method. Programs should always protect against user and system errors. Assertions are important, because they can catch programming bugs and can be removed during production.

Writing large programs with teams of programmers requires more discipline, better documentation, and better communication than one-person or small applications. Writing readable, well-documented methods is essential.

delegation	optimization	robustness
documentation	programming-in-the-large	visibility
encapsulation	refactoring	
extensibility	reusability	

Figure 20.7 Key concepts for Chapter 20

Bibliographic Notes

OO programming must render application concepts into language constructs in a correct and maintainable way, so good style is important. Most conventional programming principles apply to OO programming. [Kernighan-99] is a well-written style guide for programming.

[Brooks-95] has excellent advice for programming in the large. [Vermeulen-00] has detailed programming guidelines for Java, but many of the ideas transcend Java.

References

[Baker-95] Brenda S. Baker. On finding duplication and near-duplication in large software systems. *Second IEEE Working Conference on Reverse Engineering*. July 1995, Toronto, Ontario, 86–95.

[Brooks-95] Frederick P. Brooks, Jr. *The Mythical Man-Month, Anniversary Edition*. Boston: Addison-Wesley, 1995.

[Kernighan-99] Brian W. Kernighan, Rob Pike. *The Practice of Programming*. Boston: Addison-Wesley, 1999.

[Lieberherr-89] Karl J. Lieberherr, Arthur J. Riel. Contributions to teaching object-oriented design and programming. *OOPSLA '89* as *ACM SIGPLAN 24*, 11 (November 1989) 11–22.

[Vermeulen-00] Allan Vermeulen, Scott W. Ambler, Greg Bumgardner, Eldon Metz, Trevor Misfeldt, Jim Shur, Patrick Thompson. *The Elements of Java Style*. Cambridge, UK: Cambridge University Press, 2000.

Exercises

20.1 (4) One technique for code reuse is to use a method as an argument for another method. For example, one operation that can be performed on a binary tree is ordered printing. The subroutine *print(node)* could print the values in a tree rooted at *node* by a recursive call to *Print(node.leftSubtree)* if there is a left subtree, followed by printing *node.value*, followed by a recursive call for the right subtree. This approach could be generalized for other operations. List at least three operations that could be performed on the nodes of a binary tree. Prepare pseudocode for a subroutine *orderedVisit(node, method)* that applies *method* to the nodes of the tree rooted at *node*, in order.

20.2 (3) Combining similar methods into a single operation can improve code reuse. Revise, extend, or generalize the following two methods into a single operation. Also list the attributes needed to track both types of accounts.

a. *cashCheck (normalAccount, check)* If the amount of the *check* is less than the balance in *normalAccount*, cash the check and debit the account. Otherwise, bounce the check.

b. *cashCheck (reserveAccount, check)* If the amount of the *check* is less than the balance in *reserveAccount*, cash the check and debit the account. Otherwise, examine the reserve balance. If the check can be covered by transferring funds from the reserve without going over the reserve limit, cash the check and update the balances. Otherwise, bounce the check.

20.3 (4) Figure E20.1 is a function coded in C to create a new sheet for a computer-aided design application. A sheet is a named, displayable, two-dimensional region containing text and graphics. Several sheets may be required to completely represent a system. The function given in the fig-

```
Sheet createSheet (sheetType, rootName, suffix)
SheetType sheetType;
char *rootName, *suffix;
{ char *malloc(), *strcpy(), *strcat(), *sheetName;
  int strlen(), rootLength, suffixLength;
  Sheet sheet, vertSheetNew(), horizSheetNew();
  rootLength = strlen(rootName);
  suffixLength = strlen(suffix);
  sheetName = malloc(rootLength + suffixLength + 1);
  sheetName = strcpy(sheetName, rootName);
  sheetName = strcat(sheetName, suffix);
  switch(sheetType)
  { case VERTICAL:
      sheet = vertSheetNew();
      break;
    case HORIZONTAL:
      sheet = horizSheetNew();
      break;
  }
  sheet->name = sheetName;
  return sheet;
}
```

Figure E20.1 Function to create a new named sheet

ure creates a new vertical or horizontal sheet and constructs a name from a root and a suffix. The C functions it calls are *strlen* to compute the length of a string, *strcpy* to copy a string, *strcat* to concatenate two strings, and *malloc* to allocate memory. The data types *SheetType* and *Sheet* are defined outside of the function in the same module. The functions *strlen*, *strcpy*, and *strcat* will cause a crash if they are called with 0 for any argument. As it stands, the subroutine is exposed to several types of errors. The arguments *rootName* and *suffix* could be zero and *sheetType* could be an illegal enumerated value. The call to *malloc* could fail to allocate memory.

a. Prepare a list of all the ways the function could fail. For each way, describe the consequences.
b. Revise the function so that it does not crash as a result of any of the errors you listed in part *a* and so that it prints out a descriptive error message for each kind of error as an aid in debugging programs in which it is called.

Part 4

Software Engineering

Chapter 21	**Iterative Development**	**395**
Chapter 22	**Managing Modeling**	**403**
Chapter 23	**Legacy Systems**	**423**

At this point, you have read and hopefully have learned from the first three parts of the book. You are now familiar with OO concepts and the UML notation for expressing them. Furthermore, you have a process for applying the concepts and know how to handle the implementation details with the C++ and Java languages as well as databases. Part 4 builds on this basic knowledge and elaborates some software engineering considerations.

We have stressed throughout the book that software development should be an iterative process. Our presentation is confined by the medium of a book to be linear, but we do not want to give readers the wrong impression. Software development rarely proceeds in a straight line. Chapter 21 elaborates this theme of the iterative nature of software development.

We find that most organizations are truly interested in OO technology, and especially OO modeling. However, many organizations find it difficult to inject the technology in their ranks and are unsure how to proceed. Chapter 22 has advice for how you can capitalize on the potential of OO technology and assimilate it within your organization.

Lastly, few applications are truly new and created from scratch. In practice, new development efforts build on the experiences of predecessor applications. Legacy systems can be a rich source of requirements for their successor systems. Additional issues arise with legacy systems such as data conversion and integration of related systems. Chapter 23 touches upon these topics.

Part 4 completes the content of this book and prepares you to proceed with OO modeling and its application on your own. As always, we welcome your comments and experiences to deepen our own understanding. Please send us email (blaha@computer.org) if you have any questions or comments.

21

Iterative Development

A written presentation, such as this book, might seem to imply a linear process, but that is an unintended artifact of the medium. Software development is by its very nature iterative—early stages lack perfect foresight and must be revisited to correct errors and make improvements. ***Iterative development*** is the development of a system by a process broken into a series of steps, or iterations, each of which provides a better approximation to the desired system than the previous iteration [Rumbaugh-05].

21.1 Overview of Iterative Development

We favor an iterative approach to software development. In this chapter, we start by comparing iterative development with two other common approaches: waterfall development (Section 21.2) and rapid prototyping (Section 21.3). Then we explore the following issues of iterative development.

- Iteration scope. [21.4]
- Performing an iteration. [21.5]
- Planning the next iteration. [21.6]
- Modeling and iterative development. [21.7]
- Identifying risks. [21.8]

21.2 Iterative Development vs. Waterfall

In the 1980s and early 1990s the waterfall approach was the dominant life-cycle paradigm [Larman-03]. As Figure 21.1 shows, with this approach, developers perform the software stages in a rigid linear sequence with no backtracking. Each stage must complete before the next stage begins.

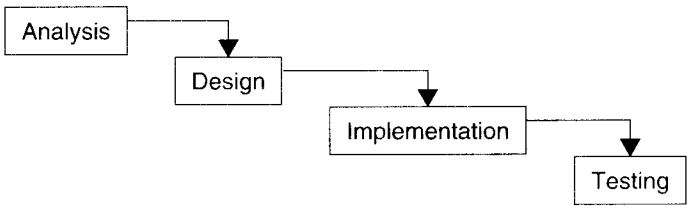


Figure 21.1 Waterfall approach. The waterfall approach is inflexible and unsuitable for most application development.

From experience, the software development community has found that the waterfall approach is not effective for building most applications [Sotirovski-01]. A waterfall is suitable for well-understood applications with predictable outputs from analysis and design, but such applications seldom occur. Most applications have substantial uncertainties in their requirements. Furthermore, a waterfall approach does not deliver a useful system until completion. This makes it difficult to assess progress and correct a project that has gone awry.

In contrast, iterative development provides frequent milestones and uncovers pitfalls early in development. When you catch difficulties early, a system is more malleable and amenable to change; revisions are easier to make and less costly than if they are deferred. Iterative development is clearly a better approach.

21.3 Iterative Development vs. Rapid Prototyping

With rapid prototyping (Figure 21.2) you quickly develop a portion of the software, use it, and evaluate it. You then incorporate what you have learned and repeat the cycle. Eventually, you deliver the final prototype as the finished application or, after a few prototypes, switch to another approach.

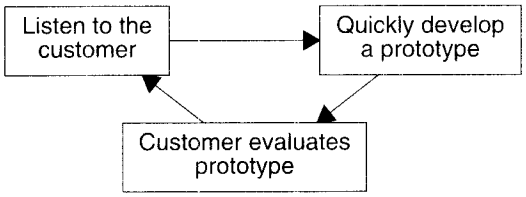


Figure 21.2 Rapid prototyping. Rapid prototyping has similar strengths to iterative development. The difference is that rapid prototyping often throws away code, while iterative development seeks to accumulate code.

Iterative development differs from rapid prototyping. Prototyping is proof of concept and often throwaway by intent. In contrast, iterative development is not throwaway; subsequent iterations build on the progress of prior ones. With iterative development, some code may be discarded due to revisions, but such throwaways are not the intent.

The strength of rapid prototyping is that it promotes communication with the customer and helps to elicit requirements. Rapid prototyping can also be helpful to demonstrate technical feasibility, where there is a potentially difficult technology. The downside of rapid prototyping is that it can be difficult to discard code. Customers often confuse a successful prototype with a product, not realizing that a prototype is just a demonstration and may lack a robust infrastructure. There is a natural reluctance to discard code; some customers regard throwing away code as throwing away money and do not realize that the true value of the prototype code is the lessons that are learned.

Iterative development has this same benefit, as long as iterations are kept small and are shown often to the customer. Both rapid prototyping and iterative development provide frequent checkpoints for assuring customers that development is going well. They also let developers resolve troublesome issues early in application development.

21.4 Iteration Scope

Iterative development consists of a series of iterations. The number of iterations and their duration depends on the size of a project. For a short project (six months or less) you might have iterations of two to four weeks. For a large multiyear project, iterations of three to four months may be more effective. If iterations are too small, the overhead of iterations is too high. If iterations are too large, there are insufficient checkpoints to assess the progress of an application and make midstream corrections. You should strive for a uniform length of iterations, but may occasionally need a longer length for deep infrastructure or difficult features.

Define the scope of an iteration—a good target is the minimum amount of work that represents material progress. Build mission-critical pieces early, as well as core pieces of code that are frequently executed by the application. Also, make sure that you balance functionality across a system. Developers will have their favorite technologies and prefer different aspects, but your overall plan must be balanced and targeted at realizing meaningful chunks of the application as quickly as possible. Each iteration must provide at least one of the following: economic payback, added functionality, improved user interaction, better efficiency, higher reliability, or strengthened infrastructure (for maintenance and future iterations).

Use cases provide a good basis for assignment. Each iteration can focus on a few use cases. However, an iteration need not complete a use case, and a use case can be spread across several iterations. For example, you might implement the core functionality in one iteration, more advanced functionality in another, and error handling in another. Don't break a use case across too many iterations, however. In addition to use cases, you must also assign internal services—mechanisms and services that provide infrastructure or support for implementing higher-level operations. These services will be identified during architectural planning and class design.

If something must be increased in priority, then something else must be decreased. This prevents the syndrome of "everything is equally important." Everything is never equally important, but managers and developers are frequently unwilling to make hard choices or to admit that there is not enough time. By maintaining the timing of iterations and adjusting their

content, you are forced to be realistic about what you can do and where you are on the schedule.

You need not release the results of each iteration to the customer. From a development perspective, it is important to maintain momentum, stay on schedule, and make sure the different components of an application actually fit together. From the customer's perspective, it may be too much effort to install each iteration. A business may combine several increments before deployment to simplify logistics.

21.5 Performing an Iteration

Each iteration must start from a common baseline and finish with a new common baseline. Developers must integrate all versions of system artifacts and check them in at the end of an iteration. This permits everybody to work with a common set of assumptions and to keep up to date with system changes. Following this rule is absolutely essential to success.

Some developers may find this rule inconvenient. It may seem more efficient to continue development of a subsystem, without having to stop and integrate it with the rest of the system. It is certainly less convenient for a development team to have to work in small pieces and frequently coordinate with others. But it is crucial to the success of the project as a whole. If teams keep to themselves for a long time, they tend to drift apart on assumptions, interfaces, and other things. When integration does occur, it can be difficult and expensive. Worse, it often happens that different subsystems have used incompatible assumptions; then changes must be backed away or hacks put in place to paper over the differences.

A team must structure its work on an iteration to be able to finish it, check it in, test it, and integrate it with the rest of the system. This requires some planning, but it pays off in the long run with the whole system.

The second rule is that each iteration must produce an executable release. It is not enough to write code that doesn't run. Code that runs can be tested. Integration of subsystems can be tested and incompatibilities discovered and corrected early. Moreover, executable code is the best measure of progress. It is very easy to delude oneself and others about the progress of design if nothing has to run. It is easy to overlook major omissions and to underestimate the difficulty of debugging and integrating subsystems.

Each iteration must include time for all of the development stages. You perform a mini-waterfall within an iteration. That is, you step through analysis, design, implementation, testing, and integration. The waterfall approach is a viable option on a small scale, permitting the systematic development of functionality. The waterfall is a problem only if decisions cannot be changed later. In the iterative process, bad decisions can be revisited in the next iteration, so they do not threaten the project.

Make sure that you plan and allow enough time for testing. It is important to test as you go, and not to defer tests until later iterations. The point of iterative development is to build a reliable system in small steps.

21.6 Planning the Next Iteration

After each iteration you should assess your progress and reconsider your plan for the next iteration. Did the iteration take more or less time than estimated? Did you have the right mix of developer skills available? Is the customer happy with the progress of the work? Have any specific problems or issues surfaced for the next iteration? Is the software stable, or must you allow for additional rework and refactoring in the next iteration?

Of course, if the prior iteration succeeded, you can continue with your plan. Otherwise, do not be afraid to discard bad decisions and make midcourse corrections. Your application will only be extensible, maintainable, and viable if it has a sound construction. You should get feedback from users early and often—you want them to internalize what you are doing and be thinking about its implications for their day-to-day business. Furthermore, they can help you detect whether the scope of the software or the path of the iterations is getting off track.

21.7 Modeling and Iterative Development

Modeling is a natural complement to iterative development. One purpose of iterative development is to discover problems in software early, and so too with modeling. [Sotirovski-01] expresses this eloquently as a "fail fast" philosophy of iterative development. Problems are inevitable, so you should root them out early. With skillful modeling, you can discover some problems in models and reduce the amount of iteration—the net result is faster and better development. Iterative development certainly is not an excuse for hacking code and foregoing the careful thought of modeling.

As Figure 21.3 shows, you should begin by carefully modeling an application to elicit requirements and then building the model for the first iteration. Then you revisit the model and do another iteration, and so on, continuing to proceed by interleaving modeling with iterative development. Modeling uncovers errors early and gives a sense of direction and continuity to a sequence of iterations. Modeling can be, and must be, done quickly so that it does not slow the project timetable.

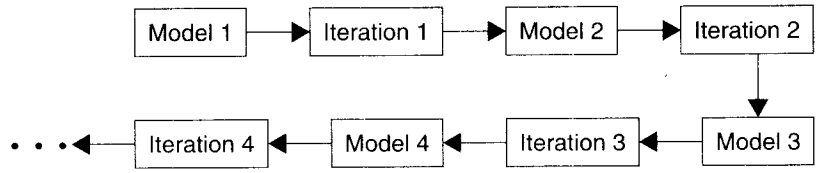


Figure 21.3 Modeling and iterative development. Modeling can improve the quality and productivity of incremental development.

Table 21.1 compares modeling with iterative development. Both promote requirements capture, but they do so in different ways. Modeling helps customers think about the potential

	Modeling	Iterative development
Requirements capture	Excellent. Models help customers understand software's capabilities and limitations before it is even built.	Excellent. Iterations show customers the software so that they can give frequent feedback.
Application quality	Excellent. Modeling fosters abstraction and deep thought.	Good. Thorough and frequent testing improves quality.
Development productivity	Excellent. The deep thought reduces rework.	Excellent. Frequent integration reduces rework.
Project tracking	Not applicable.	Excellent. Code deliveries provide frequent milestones.

Table 21.1 Comparing modeling with iterative development. Modeling and iterative development have different trade-offs. They complement each other.

of the software before it is even built. In contrast, iterative development shows customers the evolving software so that they can comment and redirect development efforts.

Modeling is unmatched in its ability to improve application quality. [Brooks-87] contends "that conceptual integrity is *the* most important consideration in system design." (The italics are his.) Modeling focuses on understanding and improving the essence of an application. Iterative development involves frequent testing, and this also contributes to quality, but the effect is less dramatic than with modeling.

Modeling improves productivity by quickly "failing" through thought experiments that preclude wasted code. Iterative development also makes a major contribution to productivity by forcing early integration, avoiding mismatched components and awkward revisions.

By definition, modeling does not deal with project tracking. Modeling concerns the early part of software development, so it is unable to have much bearing on the tracking of an entire project. Iterative development provides a great way for tracking projects—frequent deliveries of executable code leave little room for argument about what work is complete and what work remains. Consequently the schedule for a project becomes more predictable.

21.8 Identifying Risks

The key to planning an iteration is to mitigate risk. You should confront risks early, rather than defer them to the end of the project (as might otherwise occur). There are many kinds of risk to consider.

- **Technical risks.** The proposed technical solution may fail or prove unacceptable. If you address technical issues early, another solution can be found before it is too late and before the rest of the system is built on a faulty base.

- **Technology risks**. External technology that you plan to use may not be available or may not measure up to its claims. Resolve this by trying the necessary technology early in the critical parts of the system.
- **User acceptance risks**. The users may not like the user interface or the functionality of the system. Iterative development lets users try part of the system early while its style can be readily changed.
- **Schedule risks**. There is always the chance the project will not finish on time. Iterative development helps by providing an accurate measure of progress. If the schedule slips, you can trim functionality. Furthermore, even if the project finishes late, you will have an executable system to show at the deadline. A system with 90% functionality is much better than a waterfall system with 90% of the system implemented but nothing operational.
- **Personnel risks**. Key persons may leave the project at an inopportune time. Iterative development provides frequent checkpoints with delivery of a stable system. Models ensure that the iterations are carefully considered and documented. It still will be difficult to lose key personnel, but at least you have a chance of assimilating their replacements.
- **Market risks**. The requirements for an application can always change. Modeling and iterative development give you the flexibility and speed to respond.

At each iteration, you should identify risks, prioritize them, and address the highest-priority risks first. In this way, you mitigate the biggest risks early in a project, when you have time and the wherewithal to fall back to alternate approaches. Iterative plans *must* be prepared to change. This requires both a managerial climate and a working environment that understands and accepts change.

21.9 Chapter Summary

Software development is prone to miscommunication, oversights, misestimation, and unforeseen changes. Skillful modeling gives you a resilient application. Iterative development gives you a resilient process for building the application. Iterative development provides frequent milestones and uncovers pitfalls early in development.

Iterative development is different than a waterfall approach. A waterfall assumes perfect foresight and a strict development sequence. The waterfall is a failed approach that has been overemphasized in the literature.

Iterative development is also different than rapid prototyping. Rapid prototyping addresses difficult issues by exploring with throwaway code. In contrast, iterative development divides progress into small increments that intrinsically have less chance of failure. Both are valuable techniques.

The number of iterations and their duration depends on the size of a project. If iterations are too small, the overhead of iterations is too high. If iterations are too large, there are insufficient checkpoints to assess the progress of an application and make midstream correc-

tions. Several weeks or months is usually an appropriate length. Define the scope of an iteration by prioritizing risks. Attack high-priority risks first and reevaluate your priorities based on the results of each iteration. It is important not to slip the schedule of an iteration by increasing functionality once it is underway (scope creep), unless offsetting functionality is removed.

It is important that you integrate subsystems throughout development, rather than waiting to the end. If development teams keep to themselves, they tend to drift apart on assumptions and interfaces. With late integration, changes may have to be backed away or differences hacked away. Also make sure each iteration delivers an executable release and that it is tested. Executable code is the best measure of progress.

Modeling is a natural complement to iterative development. Both improve the quality, productivity, and predictability of software development. Some developers seem to think that modeling slows down development and gets in the way. But this is certainly not the case if you model deeply and quickly.

There are a number of risks that threaten development of an application. You should structure iterations to deal with the most serious risks first.

development risk	iterative development	rapid prototyping
integration	modeling	testing
iteration scope	prototype	waterfall

Figure 21.4 Key concepts for Chapter 21

Bibliographic Notes

[Larman-03] provides a thorough history of iterative development with many literature references.

References

[Brooks-87] Frederick P. Brooks, Jr. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, April 1987, 10–19.

[Larman-03] Craig Larman and Victor R. Basili. Iterative and incremental development: a brief history. *IEEE Computer*, June 2003, 47–56.

[Rumbaugh-05] James Rumbaugh. *The Unified Modeling Language Reference Manual, Second Edition*. Boston: Addison-Wesley, 2005.

[Sotirovski-01] Drasko Sotirovski. Heuristics for iterative software development. *IEEE Software*, May/June 2001, 66–73.

22

Managing Modeling

Modeling is essential for developing quality software, but it can be difficult to put into practice. Many organizations genuinely want to use models, but stumble when it comes to actually building them. Modeling requires a change in culture that an organization must actively foster.

22.1 Overview of Managing Modeling

In practice we find that many organizations are interested in using modeling but are unsure of how to proceed. This chapter provides advice for assimilating modeling into an organization and covers the following topics:

- Kinds of models. [22.2]
- Modeling pitfalls. [22.3]
- Modeling sessions. [22.4]
- Organizing personnel. [22.5]
- Learning techniques. [22.6]
- Teaching techniques. [22.7]
- Tools. [22.8]
- Estimating modeling effort. [22.9]

22.2 Kinds of Models

In practice there are several categories of models, each having different motivations, characteristics, and content. We often see practitioners confuse these different kinds of models and forget why they are building a particular model.

- **Application model**. This is the most common reason for modeling and this book's primary focus. An application model helps developers understand requirements and provides a basis for building the corresponding software. The ATM case study is an example.
- **Metamodel**. Metamodels are similar to application models but are more complex. They are often used for advanced applications. Frameworks (Chapter 14) and a class-model editor are examples.
- **Enterprise model**. An ***enterprise model*** describes an entire organization or some major aspect of it. Application models and metamodels are used for building software; enterprise models are not. Instead, enterprise models are used for reconciling concepts across applications and understanding the enterprise. By nature enterprise models have a broad scope (over multiple applications) but they are seldom deep—they need cover only the common concepts. A model of all of a bank's software would be an example.
- **Product assessment**. Models are also relevant when you are purchasing software. You should prepare multiple models—a model of your requirements and a model of the software for each of the most promising products. The requirements model is the same as an application model, except that it can lack fine detail (you are not building it). Only some vendors provide models for their products, but you can often construct your own via reverse engineering [Blaha-04]. A model helps you assess the quality of the application and understand its strengths and weaknesses as well as its scope. As an example, you might decide to buy an ATM package rather than build one, and product assessment models can clarify your decision making.

22.3 Modeling Pitfalls

The benefits of modeling are compelling, but there can be drawbacks that you should try to mitigate.

- **Analysis paralysis**. Some persons become so focused on modeling that they never finish. This situation is most likely to arise with analysts who are not developers. It can also arise with beginning modelers who are inefficient and unsure when modeling is complete.

  Resolution. A project plan can help you avoid analysis paralysis by allotting time to tasks. The plan should specify the effort for modeling and the intended deliverables. Formal reviews can give you an indication of progress and model quality. It is helpful if modelers have development experience.
- **Parallel modeling**. On several occasions, we have seen organizations construct redundant models with different paradigms. We often find OO and database modeling occurring in parallel with no communication between the respective teams. OO teams tend to be dominated by programmers who do not understand databases. Similarly, database professionals have their accustomed techniques and are often unfamiliar with OO technology. This chasm in practice mirrors the chasm in the literature. The OO and database communities have their own style and jargon, and few persons operate in both camps. The current limitations of tools exacerbate the divide.

Resolution. Oddly enough, the schism is more a matter of terminology and style, rather than substance. Your best course of action is to be aware of culture gaps. Let developers construct models for both programming and databases if they find it helpful. This is one way to cope with the limitations of current tools. In short, tolerate almost anything to get developers to model, but if there are multiple models, insist that they be frequently reconciled. Iterative development helps here—developers should reconcile at each iteration.

- **Failure to think abstractly**. Many persons cannot think abstractly and fail to learn the skill of modeling. They quickly slip into programming mode and have trouble stepping back from a problem. With models you indirectly realize an application, rather than just directly write code.

  Resolution. About the only cure is a lot of practice. Inexperienced modelers should practice solving exercises. They should work on actual applications under the guidance of a mentor. Those who are still not able to model should be assigned other tasks.

- **Excessive scope**. The purpose of modeling is to represent the real world, but only the portion relevant to your business objectives. Some people lose focus and model extraneous information. It can be reasonable to model a bit beyond your needs—after all, the exact scope of an application is seldom known up front and is partially a matter of negotiation. However, you do not want to reach way beyond application needs, because such a model is speculative and may never lead to something useful.

  Resolution. You can mitigate this pitfall with a project plan and regular reviews. Business experts need to understand what the modelers are doing, as this is their window into the capabilities and limitations of the forthcoming application.

- **Lack of documentation**. Much too often we encounter undocumented models. Diagrams alone are not sufficient; they need an explanation. A narrative should lead the reader through each diagram and define terminology. It should explain subtleties and the rationale for any controversial decisions as well as include examples to illustrate fine points.

  Resolution. You should insist on documentation, such as a model narrative or a data dictionary, and carefully read it.

- **Lack of technical reviews**. Similarly, we often encounter a lack of technical reviews. Each person or small group works on its own application in isolation and does not share experiences, knowledge, and talent. Developers do not discuss their projects, because discussion is not relevant to their immediate deliverables and there is no management encouragement. Just as business experts are a source of application requirements, development peers are a source of computing techniques and lessons from related applications. Formal reviews help to remove errors prior to testing.

  Resolution. All projects should receive at least one formal technical review, and several reviews are ideal. If there is one review, it should take place after completion of the core model and architecture. If there are several reviews, they should happen after completion of the model and architecture for each major development iteration. Management should set the tone for a critical, uninhibited, and constructive discussion. They

should make continued project funding contingent on holding a review. Keep in mind that these are *technical* reviews, not *management* reviews—the purpose of the reviews is to deepen technology, not to inform management. [Boehm-01] notes that peer reviews catch 60% of software defects, so technical reviews are clearly important. We advise that the size of technical reviews be kept small (less than ten persons) and confined to developers who are actively interested in the project.

22.4 Modeling Sessions

Chapter 12 explains domain analysis—the building of real-world models to clarify application requirements. Once you become experienced at modeling, you can consider different ways of engaging users and obtaining their input. We will characterize three alternatives—back-room, round-robin, and live modeling—and discuss their trade-offs.

22.4.1 Back-Room Modeling

The most popular way to build a model is to talk to business experts, record their comments (such as with a requirements statement or use cases), and then go off-line and model. Many analysts prefer this ***back-room modeling*** approach, because they can focus on what the user is saying and wrestle with the model later when they are alone. Over a series of meetings, users answer questions and volunteer information that they think may be helpful. After each meeting, the analyst incorporates the users' comments, and the model gradually improves. Typically, the model stays in the background and users do not see it.

It is better to meet with several users at once rather than have one-on-one meetings. A multiuser meeting has a better chemistry, because users stimulate each other's memory. Also there is a risk of intimidation in a one-on-one session, which is less likely with multiple users. Most analysts prefer to meet with a group of users who share an interest. For example, an analyst might meet separately with salespersons and engineers.

Back-room modeling has the following trade-offs.

- **Advantages**. It requires the least skill and is appropriate for analysts who are tentative with modeling.
- **Disadvantages**. The painstaking cycle of interaction with users is cumbersome for skilled modelers. The slow interaction can also be troublesome for users, because multiple interviews are required. Analysts must carefully transcribe information, or it will be forgotten.

22.4.2 Round-Robin Modeling

Round-robin modeling is more complex than back-room modeling, but more efficient at gathering requirements. The analyst still meets with small groups of users, segmented by interest or functional area, but in round-robin modeling, the users see the model. As users express requirements, the analyst traverses the model and tries to resolve them. An analyst can resolve simple issues during a meeting and complex issues afterward.

We call this approach ***round-robin modeling***, because an analyst shows the model to each group in a series of meetings until all concerns are addressed. Several iterations are required, because one group might surface an issue that an analyst needs to confirm with a previous group. Back-room modeling also parades from group to group, but users don't see the model.

We initiate round-robin modeling with a seed model that is based on existing business documentation. We don't like to start with a blank sheet of paper, because it wastes time and tries the patience of users. In contrast, a seed model stimulates discussion. Users see the analyst as well prepared and can focus on deeper issues.

In the meetings, we tell the users that they are the business experts and that we need their help in capturing requirements; we are the computer experts, and they should let us handle the details. Generally, users heave a sigh of relief. We don't dwell on formalisms and explain notation as we go. Participants don't have a problem, because we continually explain the model.

Round-robin modeling has the following trade-offs.

- **Advantages**. It requires fewer meetings than back-room modeling. Because the model is prominent, an analyst can resolve some issues during meetings. In contrast, with back-room modeling, the analyst just takes notes and may overlook needed details.
- **Disadvantages**. It still requires several iterations, and it is inefficient to shuttle ideas across the user groups. If there is contention, it can be difficult to reach agreement. The analyst is in the uncomfortable position of being an intermediary among conflicting user groups. Back-room modeling also shares this flaw. Some users do not understand models or may fear them, so the analyst must take care to allay their concerns.

22.4.3 Live Modeling

Live modeling is appropriate for expert modelers. We arrange a meeting of 10–20 persons with a range of interests—developers, managers, and various kinds of business experts. During the meeting, we build a model on the fly, listening to suggestions, volunteering comments, resolving names, and agreeing on scope. Usually, we can keep pace with the dialogue; we pause a moment when we get overloaded. A projector displays the model, which we draw with a modeling tool. A typical session lasts about two hours, and three sessions can usually elicit 80 percent of the structure for a model with 50 classes. Large and complex models take additional sessions. In between sessions, the modeler cleans up diagrams, documents the model, and resolves open issues.

The process is stimulated by the size and variety of the group; reluctant participants see others react and want to get involved to air their point of view. Comments from one person tend to trigger comments from another. We have been especially successful with skeptics.

It is acceptable to start live modeling with a clean sheet of paper, but the analyst should prepare and learn about the application in advance. The ideas will come quickly, and the analyst must be ready. Sometimes we prepare a seed model if we have prior information. Normally, we request that our client prepare a requirements statement to stimulate discussion if there are any lulls.

We are active facilitators, not just passive recorders; we ask questions and probe the attendees when answers seem unsatisfactory. We make suggestions on the basis of our experience. Ultimately, business experts make the final decisions. Occasionally, we encounter a deep modeling issue that we defer until the next meeting.

Often there are animated discussions over names. These can be helpful. Good names avoid misunderstandings. Also the discussions stimulate related information. We press business experts to devise good names—names that are brief, crisp, and not subject to confounding interpretations.

Live modeling has the following trade-offs.

- **Advantages**. This is clearly the best way to obtain user input for proficient modelers. We practice live modeling all the time, and clients are delighted by the rapid progress. The participants have different areas of knowledge and different perceptions; by working together in the same meeting, they can reconcile their views.

  A major side benefit is that the meetings induce the participants to talk to each other. Persons from different backgrounds who usually don't have the time or inclination are brought together and converse.

- **Disadvantages**. An analyst has to be highly confident of modeling, able to run a meeting, and adept with a modeling tool. Few developers have this combination of skills. Live modeling is good at eliciting structure—classes and relationships. It is less effective at finding attributes, because it is difficult to coordinate a large group for fine detail. Other input sources can provide attributes.

  Live modeling is not suitable for difficult applications, such as applications with intense metamodeling. For these situations we recommend back-room modeling.

Table 22.1 summarizes the trade-offs for the three kinds of modeling sessions.

	Back-room modeling	Round-robin modeling	Live modeling
Explanation	Record user comments and build model offline	Show model to user groups, but still build it offline	Build model during a meeting with all the users
Required skill level	Low	Medium	Very high
Productivity (for a model with 50 classes)	Low (about 15 meetings, each 2 hours long)	Medium (about 12 meetings each 2 hours long)	Very high (about 3 meetings, each 2 hours long)
Net recommendation	Best for a novice modeler	Best for a modeler with some application experience	Best for a very experienced modeler

Table 22.1 Trade-offs for different approaches to modeling sessions. Consider different approaches to modeling and their trade-offs.

22.5 Organizing Personnel

As Figure 22.1 shows, a large organization can most effectively service demand by placing a few experts in a technology-oriented group that supports groups of developers organized by business area. Table 22.2 clarifies the respective roles of the technology and application groups.

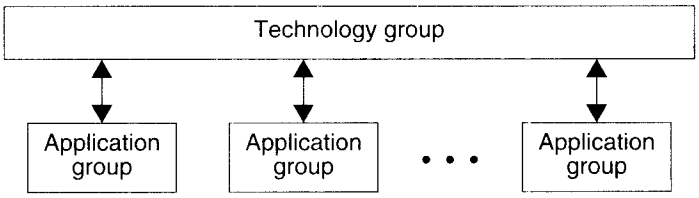


Figure 22.1 Corporate structure. A technology group can provide expertise for groups of developers organized by business area.

	Technology group	Application group
Perspective	The entire organization	A business area
Appropriate tasks	Promulgate standards and computing techniques; maintain enterprise models; support the application groups	Build applications; evaluate products for potential purchase
Required modeling skill	Expert	Fluent
Number of groups	One per organization	Many per organization
Size of group	Small to limit overhead	As many as needed to serve business area

Table 22.2 Technology vs. application groups. A large organization needs both kinds of groups.

The technology group takes the perspective of the entire organization. It promulgates standards and computing techniques and supports the application groups. You should place the best modelers and experts in the technology group, so that their skills are available to everyone. The technology group should not build applications; this is the purpose of the application groups. Rather, the technology group should be the custodian of advanced skills. Keep this group small to limit overhead.

Application groups have a different role. Their purpose is to learn about the business and transfer knowledge across related applications. Application group members should work closely with their business counterparts. Developers should be fluent with modeling, even though the best modelers belong to the technology group.

Some firms use a different organization; they place all modelers in a technology group and loan them out to perform modeling for application developers. We advise against this arrangement. Modeling is such a stimulus to insight and dialog that it should be dispersed across an entire computing organization. (For that matter, it is also beneficial if some business and marketing staff learn about models.) Modeling is the lingua franca for software development, and application developers should build models for themselves.

22.6 Learning Techniques

There are various actions that a person can take to learn about modeling. Some of these actions individuals can take on their own. Other actions require organizational support. This advice pertains to students in universities as well as practitioners in industry.

- **Training and mentoring**. Universities and commercial training houses both offer courses that explain modeling concepts and how to apply them. It is best to receive training shortly before (ideally a few weeks ahead of) its use on an actual project. Try not to receive training far in advance, or you will forget too much.

  Reinforce training with mentoring. Developers will need active help as they seek to apply the training material. Novice modelers will lack confidence that they are modeling correctly and need experience on which to draw. It will not suffice to bring in outside resources to service a project. There must be a transition of knowledge from the outside resources to in-house developers.
- **Teaming**. Application models should be constructed by small teams that initially consist of developers, business experts, and external consultants. After several applications, an organization should no longer need the external consultants, and the best in-house modelers can provide expertise. The purpose of teaming is to disseminate knowledge within a firm about both computing technology and the business.
- **Seminars**. Periodic seminars provide cost-effective education. A firm should encourage developers to present technical seminars. Seminars get developers talking and exchanging ideas. They learn about the various projects and can leverage related efforts. Seminars provide peer support for dealing with the difficulties of modeling.
- **Continual learning**. Developers should strive to find new ideas and adopt the best practices of the larger software community. Periodic attendance at technical conferences and professional meetings is helpful. Books and magazines can provide useful ideas.
- **Technical reviews**. Formal technical reviews promote conversation and become a learning experience for both the presenters and the reviewers. The reviews provide a forum for technical staff members to help one another. (See Section 22.3.)

22.7 Teaching Techniques

There are different ways to teach modeling that have various trade-offs.

- **Induction**. A person learns to model only by *doing* it, not by *talking* about it. Over the years we have tried a number of techniques and have found induction to be the quickest way to start modeling. When we run modeling sessions (see Section 22.4), we forego the preamble of a modeling tutorial (even though many attendees have never experienced a model before) and jump right into modeling the application problem. After several sessions, attendees have a good start on their application and have at the same time started to learn about modeling.
- **Practice**. Students should receive extensive hands-on practice and exercises. They should solve problems from disparate domains, with different kinds of input, different levels of abstraction, and varying difficulty.
- **Correction**. Students often make modeling mistakes, and it is difficult to anticipate all the possibilities. Part of the process of learning is for students to make mistakes and receive correction. They can learn both from their mistakes and the mistakes of their peers—through joint work with other students and class presentations. An academic setting can coerce the presentations. but in industrial courses we normally make presentations voluntary.
- **Implementation**. It is important that students understand that models can be readily implemented. They must realize that any model they can express can be implemented in a robust, predictable, and efficient manner. However, once they grasp that point, they need to set aside implementation concerns and think directly in terms of models.
- **Apprenticeship**. Another way of teaching is individually rather than en masse with a class. A new modeler can learn by forging a close working relationship with a skilled modeler. This form of teaching is best for someone who has already started to learn.
- **Patterns**. When we model, we always think in terms of patterns (Chapter 14). We encounter application situations, recognize their abstract mathematical underpinnings, and then jump to a pattern. A pattern provides a tried and true solution to a standard problem that has been studied by experts and is known to work well. There are many kinds of patterns: analysis, architecture, design, and implementation. One problem with patterns is recognizing when to apply them. Also even though they are important, patterns cover only part of a model.

22.8 Tools

Any serious software development effort requires tools—tools for modeling, configuration management, code generation, simulation, compiling, debugging, performance profiling, and so forth. We do not attempt to cover tools completely here, because there are so many kinds and such a variety of products. Instead, we focus on tools directly relevant to modeling and mention some prominent vendors.

22.8.1 Modeling Tools

Large applications (50 classes or more) require a heavyweight modeling tool. The minor benefit of a tool is that it increases productivity. The major benefit is that it can deepen think-

ing. Tools help experts build models more quickly and organize information about classes in a form that is easy to search. Tools help novices observe syntax and avoid common mistakes. IBM Rational Rose XDE, Rhapsody, Magic Draw, Together/J, and Enterprise Architect are heavyweight tools for the UML notation.

Small applications are less demanding and do not absolutely require a heavyweight modeling tool. Nevertheless, it is still a good idea to use a tool for small applications—modeling is essential to clear thinking, and the use of a tool eases model construction. Given the availability of inexpensive tools, as well as site licenses, there is seldom a good reason for not using a modeling tool.

22.8.2 Configuration Management Tools

Serious applications involve a number of files—files for programs (source code, compiled code, and executable code), documentation (for users, administrators, and maintainers), and data (configuration data, metadata, and test data). In practice it is difficult to coordinate all these files, and that is the purpose of configuration management tools. The tools improve developer efficiency and reduce the risk of losing useful work.

[Pressman-97] lists five major tasks that constitute configuration management.

- **Identification**. A configuration management tool must provide a mechanism for identifying each configuration file and relating it to other files.
- **Version control**. An organization must be able to track copies of a file as it evolves over time. Sometimes it is necessary to revert to old files (such as when trying to find the cause of a bug).
- **Change control**. An organization must determine who can approve changes as well as synchronize the work of collaborating developers. Check-in and check-out of files is a popular protocol.
- **Auditing**. A configuration management tool keeps a log of access activity. Users can access the log and determine the precise revisions to particular files, who made the changes, and the date of changes.
- **Status accounting**. There must be a means for reporting changes to others.

You can get by without configuration management for models if a single person is doing modeling and that person is disciplined about backups. However, an ad-hoc approach becomes increasingly difficult as the number of models and modelers increases. When you must manage many models and coordinate multiple persons, you should use configuration management software.

Prominent configuration management tools include IBM Rational's ClearCase, Merant PVCS, Microsoft's Visual SourceSafe, and the public-domain tool CVS.

22.8.3 Code Generators

Many modeling tools can generate application code. The typical modeling tool can generate data declarations for a program and a database, if you are using one. Some tools can generate

algorithmic code, but that is more difficult. For example, a tool might generate programming code for a state diagram.

Regardless of the tools used, developers should be careful with generated code and spot-check it for correctness and efficiency. Some tools generate bad code with flaws that are subtle and difficult to catch. Also, to our surprise, some tools have had gross errors in their outputs. If your developers pay attention to tool output, they will better understand what the tool is doing.

22.8.4 Simulation Tools

Tools can also be used to predict the behavior and performance of a finished application. Some modeling tools can simulate the performance and behavior of the finished software in advance of building it. For example, i-Logix's Statemate can simulate state diagrams.

22.8.5 Repository

Repositories are also important to application development, because they store metadata that lets the various tools communicate. A repository sits at the center of tool usage. Because they involve metadata, repositories are difficult to deploy, but effective use of a repository can leverage your usage of the individual tools. Allen System Group, Computer Associates, IBM, and Microsoft have repository products.

22.9 Estimating Modeling Effort

Any software development effort is an economic proposition. Business people estimate the cost of building the software along with the resulting revenues and cost savings. Modeling is generally a small part (much less than 10%) of the overall application effort. The following factors affect modeling effort.

- **Application complexity**. Tangible applications are simpler; highly abstract applications take longer. For example, it is easier to build software for handling customer calls than to build a system for all kinds of customer interaction.
- **Proficiency**. A skilled modeler can work an order of magnitude faster than an inexperienced one. In addition, a skilled modeler is more likely to produce a quality model with thoughtful abstractions.
- **Tools**. It helps if the developer has access to a powerful modeling tool and is skilled with it.
- **Model size**. The time to construct a model is not linear with its size. Modeling time is roughly proportional to the number of classes to the one-and-one-half power. Thus, construction for a model with 500 classes takes about 30 times longer than for one with 50.
- **Reviews**. Thorough review reduces the number of iterations needed for a model.

Given all these factors, most models require from two weeks to six months of effort.

22.10 Chapter Summary

This chapter has covered several topics to help an organization assimilate the technology of modeling.

In practice, there are several categories of models—application, meta, enterprise, and product assessment—with different motivations, characteristics, and content. Many developers overlook how to properly use modeling for the various categories. For example, an enterprise model cannot contain all the details of the covered applications, or it will become unwieldy. Models can also be used to guide purchase evaluations.

Models have many benefits, but like any technology they also entail risks. We identified some major risks and noted actions that an organization can take to mitigate them.

There are different ways that you can engage users in the process of modeling. We listed several kinds of interactions—back room, round robin, and live—along with their trade-offs.

There are various actions that a person can take to learn about modeling, including training, mentoring, teaming, seminars, continual learning, and technical reviews.

Similarly, there are different ways to teach modeling. One of the most successful is induction—a person learns to model only by doing it, not by talking about it. Teachers should give students extensive practice with models and give them the opportunity to have their mistakes corrected. Students must understand that models can be readily implemented but they should directly think in terms of models. Advanced modelers will recognize patterns and apply them.

Any serious software development effort requires tools. We listed tools relevant to modeling and gave some criteria for choosing tools. We also gave some guidelines for estimating modeling effort.

abstraction	estimation	peer support
application model	induction	product assessment
apprenticeship	live modeling	repository
back-room modeling	mentoring	round-robin modeling
code generation	metamodel	seminar
configuration management	model review	simulation tools
continuing education	modeling pitfall	teaming
documentation	modeling tool	technical review
enterprise model	pattern	training

Figure 22.2 Key concepts for Chapter 22

Bibliographic Notes

This book says little about management issues, such as project planning, project estimation, costing, metrics, personnel assignment, and team dynamics. These are all important topics, but other books cover them, such as [Pressman-97] and [Blaha-01].

In [Colwell-03] Bob Colwell recalls some of his first-hand experiences with design reviews and stresses their importance to high-quality work.

[Berndtsson-04] describes his experiences with teaching three configurations of OO analysis and design courses over a nine-year period. He presents data for the following conclusions.

- **Programming success is not an indicator of modeling success.** 65% of students who did well in an OO programming course performed poorly in an OO modeling course.
- **Modeling success is an indicator of programming success.** 84% of students who did well in an OO modeling course also did well in an OO programming course.
- **Abstraction is the key difficulty with modeling.** There is a strong correlation in grades between an OO modeling course (high abstraction) and a distributed systems course (also high abstraction).

[Box-00] concludes that OO technology involves more abstraction than the older technique of structured technology. The authors consider abstraction to be the difficult learning action.

References

[Berndtsson-04] Mikael Berndtsson. Teaching object-oriented modeling and design. *Draft paper*, 2004.

[Blaha-01] Michael R. Blaha. *A Manager's Guide to Database Technology: Building and Purchasing Better Applications*, Upper Saddle River, NJ: Prentice Hall, 2001.

[Blaha-04] Michael Blaha. A copper bullet for software quality improvement. *IEEE Computer*, February 2004, 21–25.

[Boehm-01] Barry Boehm and Victor R. Basili. Software defect reduction top 10 list. *IEEE Computer*, January 2001, 135–137.

[Box-00] Roger Box and Michael Whitelaw. Experiences when migrating from structured analysis to object-oriented modeling. *Fourth Australasian Computing Education Conference*, Melbourne, Australia, December 4–6, 2000, 12–18.

[Colwell-03] Bob Colwell. Design reviews. *IEEE Computer*, October 2003, 8–10.

[Pressman-97] Roger S. Pressman. *Software Engineering: A Practitioner's Approach, Fourth Edition*. New York: McGraw-Hill, 1997.

23

Legacy Systems

Most development does not involve new applications but rather evolves existing ones. Rarely can you build an application completely from scratch. Even if you do get to build a new application, you will often need to gather information from existing applications and integrate with them. You can salvage requirements, ideas, data, and code.

It is difficult to modify an application if you don't understand its design. If an application was previously designed using OO models and they are accurate, you can use the models to understand and evolve the application. If the models are lacking or have been lost, you should start by building a model of the existing design.

23.1 Reverse Engineering

Reverse engineering is the process of examining implementation artifacts and inferring the underlying logical intent. Reverse engineering has its origins in the analysis of hardware—where the practice of deciphering designs from finished products is commonplace [Rekoff-85]. Models facilitate reverse engineering, because they can express both abstract concepts and implementation decisions.

When building new applications, the purpose of reverse engineering is to salvage useful information. It is not intended to perpetuate past flaws. The reverse engineer must determine what to preserve and what to discard. You should regard reverse-engineered models as merely one source of requirements for a new application.

Reverse engineering requires judgment—interpretative decisions by the developer—and cannot be fully automated. Tools can assist with the sheer volume of code to be reverse engineered and the rote. However, it is difficult for the current tools to accept human decisions. Many modeling tools can generate an initial model, but this model is little more than a visual representation of the program structure. The reverse engineer must overcome at least two problems with the program code: retrieving obscure or lost information and uncovering implicit behavior.

23.1.1 Reverse Engineering vs. Forward Engineering

As Table 23.1 shows, reverse engineering is the inverse to normal development (forward engineering); you start with the actual application and work backward to deduce the requirements that spawned the software.

Forward engineering	Reverse engineering
Given requirements, develop an application.	Given an application, deduce tentative requirements.
More certain. The developer has requirements and must deliver an application that implements them.	Less certain. An implementation can yield different requirements, depending on the reverse engineer's interpretation.
Prescriptive. Developers are told how to work.	Adaptive. The reverse engineer must find out what the developer actually did.
More mature. Skilled staff readily available.	Less mature. Skilled staff sparse.
Time consuming (months to years of work).	Can be performed 10 to 100 times faster than forward engineering (days to weeks of work).
The model must be correct and complete or the application will fail.	The model can be imperfect. Salvaging partial information is still useful.

Table 23.1 Forward engineering vs. reverse engineering. Reverse engineering is the opposite of forward engineering and requires a different mindset.

23.1.2 Inputs to Reverse Engineering

When performing reverse engineering, you must be resourceful and consider all inputs. The available information varies widely across problems.

- **Programming code**. The programming source code can be a rich information source. Tools can help you understand the flow of control and the data structure. Comments and suggestive names of variables, functions, and methods can deepen your understanding.
- **Database structure**. If the application has a database, you can also learn from it. The database specifies the data structure and many constraints—precisely and explicitly.
- **Data**. If data are available, you can discover much of the data structure. A thorough application program or disciplined users may yield data of better quality than the data structure enforces. For large systems, you may have to sample the data to reach tentative conclusions, and then explore further for verification. Examination cannot prove many propositions, but the more data you encounter, the more likely will be the conclusion.
- **Forms and reports**. Suggestive titles and layouts can clarify data structure and processing logic. Form and report definitions are especially helpful if their binding to variables

is available. An empirical approach is to enter known, unusual values to establish the binding between forms and the underlying variables.

- **Documentation**. Problems vary in their quality, quantity, and kind of documentation. Documentation provides context for reverse engineering. User manuals are especially helpful. Data dictionaries—lists of important entities and their definitions—may be available. Be careful with all documentation, because it can become stale and inconsistent with application code.
- **Application understanding**. If you understand an application well, you can make better inferences. Application experts may be available to answer questions and explain rationale. You may be able to leverage models from related applications.
- **Test cases**. Test cases are intended to exercise the normal flow of control and unusual situations. Sometimes they provide useful clues.

23.1.3 Outputs from Reverse Engineering

Reverse engineering has several useful outputs.

- **Models**. The model conveys the software's scope and intent. It provides a basis for understanding the original software and building any successor software.
- **Mappings**.You can tie model attributes to variables. Less precisely, you can bind programming code to state and interaction models.
- **Logs**. Reverse engineers should record their observations and pending questions. A log documents decisions and rationale.

23.2 Building the Class Model

Begin by constructing a class model of the application so that you can understand the classes and relationships. We suggest building the class model using three distinct phases: implementation recovery, design recovery, and analysis recovery.

23.2.1 Implementation Recovery

First quickly learn about the application and create an initial class model. If the program is written in an OO language, you can recover classes and generalizations directly. Otherwise you must study the data structures and operations and manually determine classes. The system may lack a proper design, so the result may not be pleasing. Try to avoid making any inferences other than determining classes at this point. It is helpful to have an initial model focused on the implementation.

23.2.2 Design Recovery

Next probe the application to recover associations. The typical implementation of an association is as a single pointer attribute. The multiplicity in the forward direction is usually clear.

In contrast, the multiplicity in the reverse direction is typically not declared, and you must determine it from general knowledge or examination of the code.

Many implementations use a collection of pointers to implement an association with "many" multiplicity. Then the initial generated model points to the collection class, rather than the class of the elements. You should move the association to the class of the elements and adjust the multiplicity accordingly. Collection classes are mechanisms and should not appear in most analysis or design models. You can mark them on associations as recommended implementation for a "many" direction.

Sometimes pointers will implement both association directions. In those cases, you must identify the matching pointers and consolidate them. You should suspect any two classes that have pointer attributes to each other. If you reverse engineer the initial model with a tool, it will have one association for each pointer. You should remove one association and move its information to the reverse direction of the other.

Multiplicity typically has a lower limit of 0 or 1. The lower limit is '0' if the target object is initialized somewhere in the source code, but it is uncertain when initialization will happen. The lower limit is '1' if the target object is initialized at object creation time, such as in the constructor or the class initialization block.

23.2.3 Analysis Recovery

Finally, you interpret the model, refine it, and make it more abstract. Remove any remaining design artifacts and eliminate any errors. Be sure to get rid of all redundant information or mark it as such. It's also a good time to reconsider the model. Is it readable and coherent? Reconcile the reverse engineering results with models of other applications and documentation. Show your model to application experts and incorporate their advice.

If your source code is not object oriented, you will have to infer generalizations by recognizing similarities and differences in structure and behavior. Similarly, you will have to use your application understanding and careful study of the code to determine aggregations and compositions. For example, objects with coincident lifetimes suggest composition. You will also need to understand the application and code to determine qualifiers and association classes.

You can add packages to organize the classes, associations, and generalizations. You can combine the classes in several code files into one package or split a large code file across multiple packages.

23.3 Building the Interaction Model

The purpose of each method is usually clear enough, but the way that objects interact to carry out the purposes of the system is often hard to understand from the code. The problem is that code is inherently reductionist: it describes how each piece works by itself. But the meaning of the system as a whole is holistic: the emergent interactions among objects give it meaning. The interaction model can give you a broad understanding.

You can add methods to the class model by using slicing. A ***slice*** is a subset of a program that preserves a specified projection of its behavior [Weiser-84]. You can perform slicing by marking some initial code to retain. Then, recursively, mark all statements used by the retained code. The accumulated code lets you project an excerpt of behavior from the original program. Thus slicing provides a means for converting procedural code into an OO representation that is centered about objects.

Programmers naturally think in terms of slicing as [Weiser-82] notes. The power of slices comes from four facts: 1) they can be found automatically, 2) slices are generally smaller than the program from which they originated, 3) they execute independently of one another, and 4) each reproduces exactly a projection of the original program's behavior [Weiser-84].

You can use an activity diagram to represent an extracted method so that you can understand the sequence of processing and the flow of data to various objects. You can then construct sequence diagrams from the activity diagrams for simplification.

23.4 Building the State Model

If you are studying a user interface, a state model can be quite helpful. Otherwise, state models are not prominent in most other kinds of application code.

If you do need to construct a state model, you can proceed as follows. As an input, you have sequence diagrams from building the interaction model. You need to fold the various sequence diagrams for a class together, by sequencing events and adding conditionals and loops as Chapter 13 describes.

You can augment the information in the sequence diagrams by studying the code and doing dynamic testing. It is helpful to find all the possible states for each class that has a state model. Initiation and termination correspond to construction and destruction of objects.

23.5 Reverse Engineering Tips

As you perform reverse engineering and build class, interaction, and state models, it will help if you keep in mind the following tips.

- **Distinguish suppositions from facts**. Reverse engineering yields hypotheses. You must thoroughly understand the application before reaching firm conclusions. As reverse engineering proceeds, you may need to revisit some of your earlier decisions and change them.
- **Use a flexible process**. We adjust the reverse engineering process to fit the problem. Problem styles and the available inputs vary widely. You must use your wits for reverse engineering. You are solving a large puzzle.
- **Expect multiple interpretations**. There is no single answer as in forward engineering. Alternative interpretations can yield different models. The more information that is available, the less judgments should vary among reverse engineers.

- **Don't be discouraged by approximate results**. It is worth a modest amount of time to extract 80 percent of an application's meaning. You can use the typical forward engineering techniques (such as interviewing knowledgeable users) to obtain the remaining 20 percent. Many people find this lack of perfection uncomfortable, because it is a paradigm shift from forward engineering.
- **Expect odd constructs**. Developers, even the experts, occasionally use uncommon constructs. In some cases, you won't be able to produce a complete, accurate model, because that model never existed.
- **Watch for a consistent style**. Software is typically designed using a consistent strategy, including consistent violations of good design practice. Usually, you can look at an excerpt of the software and deduce the underlying strategy.

23.6 Wrapping

Some applications are brittle and poorly understood—they may have been written long ago, have missing documentation, and lack guidance from the original developers. Changes can threaten their viability and risk introducing bugs. Consequently, many organizations limit changes to such applications. They prefer to isolate the code and build a wrapper around it.

A ***wrapper*** is a collection of interfaces that control access to a system. It consists of a set of boundary classes that provide the interfaces, and it should be designed to follow good OO principles. The boundary classes' methods call the existing system using the existing operations. A boundary method may involve several existing operations and bundle data from several places. Often the calls to legacy code are messy, but the boundary classes hide the details from the outside. The source code can either be OO or non-OO. Wrapping preserves the form of legacy software and accesses its functionality.

Many existing applications have functionality that is confusing, unpredictable, or complex. Often it is possible to extract a core of functionality that is more fundamental, simpler to use, and better tested. In this case, a wrapper provides a clean interface for exposing the core functionality. Some features of the original application are lost, but they are usually the most dubious ones.

If you are adding new functionality, it can usually be added as a separate package. Design this package using good OO principles. Try to minimize interactions with the existing system, and keep them as uniform as possible.

For an example consider a Web application. The original code may be a legacy banking application written in Cobol and running on old hardware. Wrapping can expose the Cobol logic as OO methods that can then be attached to a modern Web interface. The legacy code still must be maintained, but its maintenance is little affected by the existence of the Web interface. The actual code that executes for the Web application is ugly, but it works as long as maintenance on the underlying Cobol logic does not disrupt the wrapper's methods.

Wrapping is usually just a temporary solution, because a wrapper is heavily constrained by the organization (often accidental) of the legacy software. Eventually the combination of

the original old code, the wrapper, and new code in a different format become so unwieldy that it must be rewritten.

Sneed suggests the use of XML as a gateway for communication between the legacy software on the inside and the modern world on the outside. Programmers with different levels of sophistication and modernity can interact via the intermediary of XML [Sneed-01]. He also observes that wrapping serves both a technological and a social purpose. The maintainers of the old code do not have their artifacts disrupted with wrapping. Wrapping lets program maintainers keep their mental models of the software intact. This helps with the day-to-day maintenance of the wrapped code.

23.7 Maintenance

Much of the software literature treats maintenance as being monolithic, but we think the viewpoint of Rajlich and Bennett is more perceptive. According to [Rajlich-00], software moves through five stages.

- **Initial development**. Developers create the software.
- **Evolution**. The software undergoes major changes in functionality and architecture. Refactoring can be used to maintain software quality.
- **Servicing**. The available technical talent has been reduced, either by circumstances or by deliberate decision. Software changes are limited to minor fixes and simple functionality changes. At this stage the software begins its inexorable slide to obsolescence. Wrapping becomes an appropriate technology during this stage.
- **Phaseout**. The vendor continues to receive revenue from the product but is now planning its demise.
- **Closedown**. The product is removed from the market, and customers are redirected to other software.

These five stages do not have a rigid wall between them, but there is a continual decline in the technical quality of the software as it proceeds through its lifetime. Also the authors note that individual versions of software can experience these life stages, only to be replaced by successor versions.

The software engineering goal is to slow the decline and keep software in the evolution and servicing stages as long as possible. In any case, management wants to avoid an accidental slippage of the software and only let the transitions proceed with forethought.

23.8 Chapter Summary

Most development does not involve new applications but rather evolves existing ones. Accordingly, as a software engineer you must be able to evolve existing applications and integrate with them. You can salvage requirements, ideas, data, and code from existing

applications. Reverse engineering is a critical technology when dealing with legacy applications.

The purpose of reverse engineering is to salvage information from old systems and carry it forward. Reverse engineering is *not* intended to perpetuate past flaws—you discard any flaws that you find. Reverse engineering provides merely one source of requirements for new applications, but it is an important source. There can be a variety of inputs to reverse engineering, all of which you should be prepared to exploit. The primary outputs from reverse engineering are models.

Begin by building the class model, emphasizing classes, associations, and generalizations. We suggest building the class model using three distinct phases—implementation recovery, design recovery, and analysis recovery—that involve increasing amounts of decisions and interpretation about the software.

Next build the interaction model, being sure to tie your understanding of behavior to the class model. You can start with procedural code and use slicing to extract portions of the logic that are centered about objects. Hence slicing provides a means for taking the content of procedural code and restructuring it as OO code. Ultimately you express the interaction model as a collection of activity and sequence diagrams.

Finally, if you need it, build the state model. The sequence diagrams provide a helpful intermediary to state diagrams.

Reverse engineering is much different than forward engineering and consequently requires a different mindset. We provided a number of tips to help with the changed mindset.

Wrapping is another technique for dealing with legacy applications. You can regard the legacy application as a black box and build interfaces around it. New applications then access the legacy logic via the intermediary of the wrapper.

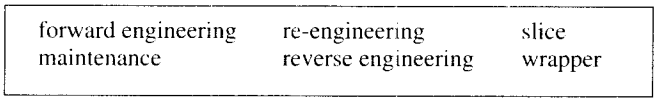


Figure 23.1 Key concepts for Chapter 23

Bibliographic Notes

[Bachman-89] explains that most information systems build on past work and only the occasional project truly involves new work. We note that this claim applies not only to information systems but also to software in general.

[Chikofsky-90] has been influential in standardizing reverse engineering terminology. [Kollmann-01] explains how to recover a class model from source code and test executions. The test executions often do not prove hypotheses about the class model, but they can help understanding and are suggestive. [Sneed-96] presents a basic approach to taking a procedural COBOL program and converting it to an OO representation.

References

[Bachman-89] Charles W. Bachman. A personal chronicle: Creating better information systems, with some guiding principles. *IEEE Transactions on Knowledge and Data Engineering 1*, 1 (March 1989), 17–32.

[Chikofsky-90] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, January 1990, 13–17.

[Kollmann-01] Ralf Kollmann and Martin Gogolla. Application of UML associations and their adornments in design recovery. *IEEE Eighth Working Conference on Reverse Engineering*, October 2001, Stuttgart, Germany, 81–90.

[Rajlich-00] Vaclav T. Rajlich and Keith H. Bennett. A staged model for the software life cycle. *IEEE Computer*, July 2000, 66–71.

[Rekoff-85] MG Rekoff, Jr. On Reverse Engineering. *IEEE Transactions on Systems, Man, and Cybernetics SMC-15*, 2 (March/April 1985), 244–252.

[Sneed-96] Harry M. Sneed. Object-oriented COBOL recycling. *IEEE Third Working Conference on Reverse Engineering*, November 1996, Monterey, CA, 169–178.

[Sneed-01] Harry M. Sneed. Wrapping legacy COBOL programs behind an XML interface. *IEEE Eighth Working Conference on Reverse Engineering*, October 2001, Stuttgart, Germany, 189–197.

[Weiser-82] M. Weiser. Programmers use slices when debugging. *Communications of the ACM 25*, 7 (July 1982), 446–452.

[Weiser-84] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering 10*, 4 (July 1984), 352–357.