5부 RAD 데이터베이스 애플리케이션 개발

31장 재고 관리 애플리케이션 - 클라이언트/서버 개발

32장 DDG 버그 보고용 도구 - 데스크탑 애플리케이션 개발

33장 DDG 버그 보고용 도구 - 웹 브로커 활용

31장 재고 관리 애플리케이션 - 클라이언트/서버 개발

* 백 엔드의 설계

* 데이터베이스 조작의 중앙 집중화 - 비즈니스 규칙

* 사용자 인터페이스의 설계

이번 장에서는 28장, “인터베이스를 이용한 클라이언트/서버 애플리케이션의 개발”에서 이야기했던 개념들을 사용해서 데이터베이스 애플리케이션을 만들어 보겠다. 이번 장을 통해서 2-티어(two-tier) 구조 클라이언트 서버를 개발하는 기법들을 배울 수 있을 것이다. 이번 예제 애플리케이션은 클라이언트와 서버 그리고 비즈니스 규칙(business rule)이라는 세 개의 티어로 나뉘어진다. 또한, 하나의 데이터 모듈을 통해서 데이터 조작을 중앙 집중화하는 기법도 배울 수 있는데, 데이터 조작을 중앙 집중화한다는 것은 사용자 인터페이스 요소를 데이터베이스 조작 요소와 완전히 분리시킨다는 뜻이다.

4장, “애플리케이션 프레임웍과 설계 개념들”에서 독립적으로 생성될 수도 있고, 다른 컨트롤의 자식 윈도우로도 생성될 수 있는 폼들에 대한 프레임웍을 소개했었다. 이번 장의 사용자 인터페이스도 그 프레임웍을 이용한다.

데이터베이스 백 엔드는 Local InterBase를 사용한다. 이번 예제는 전형적인 자동차 부품 관리 업무 모델에 관련된 것이다. 이 업무 모델의 관리해야 할 데이터는 다음과 같다.

- 제품 재고 - 각 부품들의 수량, 가격이 포함된다.
- 판매 - 판매된 제품에 대한 정보와 그 제품을 구매한 고객에 대한 정보가 포함된다.
- 고객 - 고객의 이름이나 주소 등의 정보가 포함된다.

전형적인 재고 관리 애플리케이션에 적용되는 데이터 모델이라고 할 수 있다. 이번 장의 목적은 클라이언트/서버 개발 기법을 보여주는 것이다. 그런 목적에 맞으면서도 완벽하게 작동하는 애플리케이션을 제시하겠다.

이번 장은 세 부분으로 나뉘어진다. 첫 번째 부분인 “백 엔드의 설계”에서는 백 엔드의 설계에 대해서 이야기한다. 이 부분은 28장에서 나왔던 데이터베이스 객체들이 포함된다. 두 번째 부분인 “데이터베이스 조작의 중앙 집중화 - 비즈니스 규칙”에서는 델파이의 TDataModule을 이용해서 데이터베이스 조작을 중앙 집중화하는 방법을 설명한다. 그리고 세 번째 부분인 “사용자 인터페이스의 설계”에서는 재고 관리 애플리케이션의 실제 사용자 인터페이스 설계에 대해서 이야기한다.

백 엔드의 설계

재고 관리 애플리케이션의 백 엔드로는 Local InterBase 서버를 사용한다. Local InterBase를 사용해서 얻을 수 있는 이점은, 우선 데이터베이스 전체를 SQL로 설계할 수 있다는 것이다. 그리고 트리거(trigger)나 생성기(generator), 저장 프로시저(stored procedure) 등을 통해서 데이터 처리 과정의 일부분을 서버 쪽에 옮길 수 있는 유연성이 생긴다. 결과적으로 보면 데이터의 무결성을 보장하는 데에도 유리하다. SQL 백 엔드의 또 다른 이점이라면 진정한 클라이언트/서버 환경의 다양한 상황을 충분히 처리할 수 있는 능력(scalability)을 가지고 있다는 점이다.



이번 장의 몇몇 내용들은 InterBase에만 적용되며, 오라클이나 마이크로소프트 SQL 같은 다른 SQL에는 적용되지 않는다. SQL 서버들이 사용하는 개념은 대부분 비슷하지만, 실제 구현은 조금씩 다르다는 점을 주의하기 바란다.

이번 재고 관리 애플리케이션의 여러 가지 데이터베이스 객체들은 SQL을 통해서 생성된다. SQL로 생성하는 객체들에는 도메인, 테이블, 생성기, 트리거, 저장 프로시저, 권한 등이 포함된다.

소프트웨어 시장에는 백 엔드 설계를 도와주는 다양한 데이터 모델링 도구들이 나와 있다. RoboCase나 Erwin, SQL-Designer, Rational Rose 같은 도구들을 들어보았거나 실제로 사용하는 독자들도 있을 것이다. 그러한 도구들은 대부분 SQL 코드를 직접 입력할 필요없이 시각적으로 데이터를 모델링할 수 있도록 도와주는 것이다. 또한, 기본적인 데이터 모델을 설계한 후에라도 언제든지 모델을 수정할 수 있다.

그림 31.1은 재고 관리 애플리케이션의 데이터 모델을 나타낸 것이다.

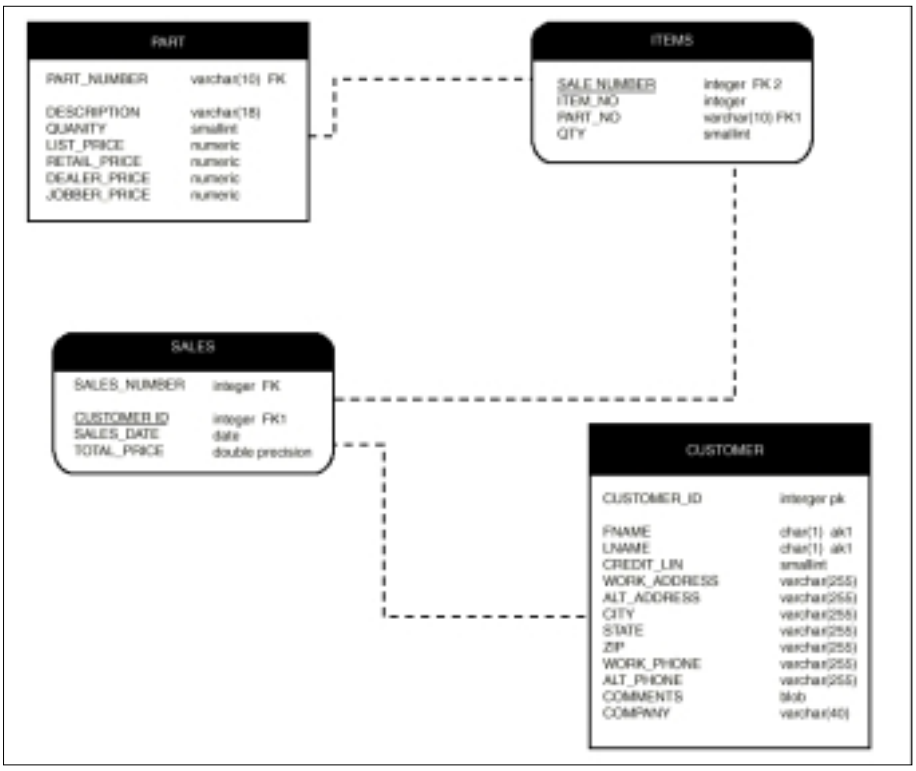


그림 31.1
재고 관리 응용 프로그램의 데이터 모델

도메인 정의

테이블이나 트리거 같은 메타데이터 객체들을 정의하기 전에 그런 객체들을 정의할 때 사용할 도메인(domain)부터 정의하는 것이 올바른 순서이다.



메타데이터(metadata)란, 데이터베이스 정의를 구성하는 모든 객체들(테이블, 색인 등등)을 뜻한다.

도메인이란, SQL의 사용자 정의 데이터형이라고 생각하면 된다. 도메인을 통해서 기본 데이터형보다 좀더 구조적인 형태의 데이터형을 정의할 수 있다.

도메인을 사용하면 SQL 코드를 좀더 읽기 쉬운 형태로 만들 수 있을 뿐만 아니라, 데이터베이스의 개별 객체들이 사용하는 데이터형을 일관적으로 관리하기에도 편하다. 그러나 테이블 필드의 정의에 일단 한 번이라도 사용된 도메인의 정의는 나중에 다시 바꿀 수 없다는 점을 주의해야 한다.

다음은 이번 예제 데이터베이스가 사용하는 몇 가지 도메인들이다.

```
CREATE DOMAIN DCUSTOMERID AS INTEGER;
```

아주 간단한 도메인이다. 그냥 INTEGER라는 이름 대신 DCUSTOMERID라는 이름을 사용하는 정수형을 만들었다고 생각하면 된다.

```
CREATE DOMAIN DCREDITLINE AS SMALLINT default 0 CHECK (VALUE BETWEEN
0 AND 3000);
```

이것은 새로운 SMALLINT형의 도메인을 정의하는데, 이 도메인으로 정의된 필드에 사용 가능한 값은 0에서 3000 사이로 제한된다.

```
CREATE DOMAIN DNAME AS CHAR(20);
```

이것은 DNAME이라는 20자리 고정 길이 문자열 도메인을 정의한다.

```
CREATE DOMAIN DADDRESS AS VARCHAR(50);[sr]
CREATE DOMAIN DCITY AS VARCHAR(20);[sr]
CREATE DOMAIN DSTATE AS VARCHAR(20);[sr]
CREATE DOMAIN DZIP AS VARCHAR(10);[sr]
CREATE DOMAIN DPHONE AS VARCHAR(20);
```

위의 정의문들은 각각 50, 20, 20, 10, 20자리 가변 길이 문자열 도메인을 정의한다.

```
CREATE DOMAIN DPRICE AS NUMERIC(15, 2) default 0.00;
```

이것은 실수를 뜻하는 도메인인데, 첫 번째 숫자인 15는 유효 자리의 개수가 15개라는 뜻이며, 두 번째 숫자인 2는 소수점 자리가 2자리라는 뜻이다. 이 도메인을 사용하는 필드의 기본값은 0.00이 된다.



CHAR(n) 데이터형은 데이터베이스에 항상 n개의 문자들을 저장한다. 필드에 설정된 문자열이 n개 이하의 문자들로 되어 있으면 나머지는 모두 빈 칸(space)으로 채워진다.
VARCHAR(n) 데이터형은 문자열의 실제 크기만큼의 문자들만 데이터베이스에 저장한다. 저장할 수 있는 문자들의 최대 개수는 n이다. 이 데이터형은 CHAR(n)보다 저장 효율은 좋지만, 속도는 조금 느리다.

도메인에 대한 좀더 자세한 정보는 Interbase Language Reference Guide나 IB32.hlp 도움말을 참고하기 바란다.

테이블 정의

도메인을 정의한 다음에는 테이블을 정의한다. 각각의 테이블은 CREATE TABLE이라는 SQL문으로 정의하며, 테이블의 각 필드들은 CREATE TABLE의 괄호 안에서 정의한다.

CUSTOMER 테이블

CUSTOMER 테이블은 고객 데이터 객체를 뜻하는 것이다. 이 테이블의 정의는 다음과 같다.

```
/* 테이블: CUSTOMER, 소유자: SYSDBA */
CREATE TABLE CUSTOMER (CUSTOMER_ID INTEGER NOT NULL,
                FNAME DNAME NOT NULL,
                LNAME DNAME NOT NULL,
                CREDIT_LINE DCREDITLINE NOT NULL,
```

```
                WORK_ADDRESS DADDRESS,
                ALT_ADDRESS DADDRESS,
                CITY DCITY,
                STATE DSTATE,
                ZIP DZIP,
                WORK_PHONE DPHONE,
                ALT_PHONE DPHONE,
                COMMENTS BLOB SUB_TYPE TEXT SEGMENT SIZE 80,
                COMPANY VARCHAR(40),
CONSTRAINT PCUSTOMER_ID PRIMARY KEY (CUSTOMER_ID));
```

NOT NULL이 붙은 필드들은 반드시 값을 입력해야 하는 필드들이다. 다시 말해서 레코드를 테이블에 보내기(post) 전에는 반드시 필드에 유효한 값을 넣어야 한다.

COMMENTS 필드는 조금 설명이 필요할 것 같다. 이 필드는 BLOB(Binary Large Object)형이다. BLOB는 크기가 큰 비정형 데이터를 저장하기 위한 것이다. SUB_TYPE TEXT의 의미는 BLOB에 저장되는 값이 ASCII 텍스트라는 뜻이며, 따라서 이 필드의 값은 델파이의 TDBMemo 컴포넌트와 호환된다.

CONSTRAINT문은 CUSTOMER_ID 필드를 테이블의 기본키로 정의한다. 따라서, 각 레코드의 CUSTOMER_ID 필드는 테이블 전체에서 고유한 값을 가져야 한다. 그리고 이 키는 데이터베이스 전체의 참조 무결성을 보장하는 역할도 담당하게 된다. 이후에 다시 나오겠지만, PRIMARY KEY로 정의된 필드는 다른 테이블에 정의된 FOREIGN KEY 필드들의 참조 필드 역할을 한다.

PART 테이블

PART 테이블은 고유한 각 부품들의 정보가 저장되는 테이블이다. 이 테이블의 정의는 다음과 같다.

```
/* 테이블: PART, 소유자: SYSDBA */
CREATE TABLE PART (PART_NUMBER VARCHAR(10) NOT NULL,
                DESCRIPTION VARCHAR(18),
                QUANTITY SMALLINT NOT NULL,
                LIST_PRICE DPRICE NOT NULL,
                RETAIL_PRICE DPRICE NOT NULL,
```

```
                DEALER_PRICE DPRICE NOT NULL,
                JOBBER_PRICE DPRICE NOT NULL,
                CONSTRAINT PPART_NUMBER PRIMARY KEY (PART_NUMBER));
```

이 테이블의 각 레코드는 각각의 부품을 뜻한다. 각 레코드에는 그 부품에 대한 설명과 수량, 가격 정보가 포함된다. 그리고 이 테이블 역시 PART_NUMBER라는 기본키가 있다.

SALES 테이블

SALES 테이블은 고객에 대한 판매 정보를 담고 있는 테이블이다. 이 테이블의 정의는 다음과 같다.

```
/* 테이블: SALES, 소유자: SYSDBA */
CREATE TABLE SALES (SALE_NUMBER INTEGER,
                CUSTOMER_ID INTEGER,
                SALE_DATE DATE,
                TOTAL_PRICE DOUBLE PRECISION);

ALTER TABLE SALES ADD FOREIGN KEY (CUSTOMER_ID)
                REFERENCES CUSTOMER(CUSTOMER_ID);
```

ALTER TABLE문에서 SALES 테이블에 대한 외부키(foreign key)를 정의한다. 외부키란 다른 어떤 테이블의 기본키로 정의되어 있는 하나의 필드 또는 일련의 필드들과 정확히 같은 순서로 대응하는 하나의 필드 또는 일련의 필드들을 뜻한다. SALES 테이블에서 CUSTOMER_ID를 외부키로 정의하면 SALES 테이블의 각 레코드들의 CUSTOMER_ID 필드값은 CUSTOMER 테이블에 존재하는 각 레코드들의 CUSTOMER_ID 필드값에 일대일로 대응하게 된다. 결과적으로, SALES 테이블과 CUSTOMER 테이블 사이의 참조 무결성(reference integrity)이 보장되는 것이다.

ITEMS 테이블

ITEMS 테이블은 특정 판매에 대한 정보가 저장되는 곳이다. SALES 테이블과 ITEMS 테이블에는 일대다(一 對 多) 관계가 성립하며, 두 테이블은 SALE_NUMBER 필드와 SALE_NO 필드를 통해서 연결된다. ITEMS 테이블의 정의는 다음과 같다.

```
/* 테이블: ITEMS, 소유자: SYSDBA */
CREATE TABLE ITEMS (SALE_NUMBER INTEGER,
                ITEM_NO INTEGER,
                PART_NO VARCHAR(10),
                QTY SMALLINT);

ALTER TABLE ITEMS ADD FOREIGN KEY (PART_NO)
                REFERENCES PART(PART_NUMBER);
```

SALES 테이블에서와 마찬가지로 PARTS 테이블에 존재하지 않는 PART_NO 필드를 가지는 레코드가 ITEMS 테이블에 입력되지 않도록 하기 위해 외부키 하나를 정의했다.

생성기의 정의

생성기(generator)란, 테이블에 삽입될 일련 번호를 자동적으로 생성하는 메커니즘이라고 생각하면 된다. 일반적으로, 테이블의 키 필드에 들어갈 고유 번호를 만들어낼 때 생성기를 사용한다. SALES 데이터베이스는 생성기를 사용해서 CUSTOMER, SALES, ITEMS 테이블에 들어갈 고유한 ID들을 만들어낸다. 각 생성기들은 다음과 같이 정의한다.

```
CREATE GENERATOR GEN_CUSTID;
CREATE GENERATOR GEN_ITEMNO;
CREATE GENERATOR GEN_SALENO;
```



데이터베이스에 생성기를 추가한 후에는 그것을 제거하기가 쉽지 않다. 가장 간단한 기법은 GEN ID()를 호출하지 않도록 트리거나 저장 프로시저를 수정, 삭제하는 것이다. 또는, RDB$GENERATORS 시스템 테이블에서 생성기를 직접 삭제할 수도 있다.

트리거의 정의

트리거(trigger)란, 테이블의 레코드가 삽입, 수정, 삭제되면 자동적으로 수행되는 루틴을 뜻한다. 트리거를 이용하면 애플리케이션의 힘을 빌리지 않고도 반복적인 레코드 조작을 수행할 수 있다.

이번 장에 나온 트리거와 생성기에 대한 내용은 InterBase에만 국한된 것이다. 다른 SQL 시스템들도 트리거나 생성기라는 개념을 지원하지만, 문법이나 구조는 조금씩 다를 수 있다. 트리거나 생성기가 편리하기는 하지만, 그러나 트리거나 생성기에 크게 의존하는 InterBase 데이터베이스를 나중에 다른 SQL 서버에 이식하려면 이식 작업이 아주 번거로워질 수 있다는 점을 염두에 두어야 한다.

우선, 앞에서 이야기한 생성기를 이용해서 고객 ID와 항목 번호를 각각의 해당 테이블에 추가하는 트리거가 필요하다. 새로운 고객 ID를 추가하는 트리거는 다음과 같다.

```
CREATE TRIGGER TCUSTOMER_ID FOR CUSTOMER
ACTIVE BEFORE INSERT POSITION 0
as begin
  new.customer_id = gen_id(gen_custid, 1);
end
```

다음은 ITEMS 테이블에 대한 트리거이다.

```
CREATE TRIGGER TITEM_NO FOR ITEMS
ACTIVE BEFORE INSERT POSITION 0
as begin
  new.item_no = gen_id(gen_itemno, 1);
end
```



위의 두 트리거 이외에도 이 데이터베이스에는 두 글자의 주 약자(CA 등)를 주 이름(California 등)으로 변환하는 트리거들이 있다.

저장 프로시저의 정의

저장 프로시저(stored procedure)란, 데이터베이스의 메타데이터의 일종으로 서버에 미리 정의되어 있는 SQL 루틴이라고 생각하면 된다.

저장 프로시저를 호출하면 저장 프로시저는 데이터셋을 돌려주는데, 이는 일반적인 SQL 질의를 수행한 결과와 다르지 않다. 저장 프로시저의 장점이라고 한다면 클라이언트 쪽에 필요한 처리량이나 네트웍 부담을 줄일 수 있다는 점을 들 수 있다. 그리고 특정한 기능을 중앙 집중적으로 수행/제어할 수 있다는 것도 중요한 장점이다. 저장 프로시저는 서버쪽에서 미리 전처리가 끝난 상태의 SQL 코드이기 때문에 일반적인 SQL 질의보다 수행 속도도 빠르고 네트웍 부담도 적다. 저장 프로시저의 여러 가지 기능에 대해서는 28장에서 자세히 설명했었다.

SALES 데이터베이스는 두 개의 저장 프로시저를 사용한다. 첫 번째 저장 프로시저인 INSERT_SALE은 SALES 테이블에 판매 레코드를 삽입하는 역할을 한다. 이 저장 프로시저는 고객 ID와 판매 데이터, 그리고 판매 총액을 인자로 받아서 새 판매 레코드를 삽입한 후 그 레코드의 판매 번호를 돌려준다. 클라이언트 애플리케이션은 INSERT_SALE이 돌려준 판매 ID를 두 번째 저장 프로시저에 넘기는데, 그 저장 프로시저는 판매 번호를 ITEMS 테이블의 외부키로 사용해서 SALES 테이블을 참조한다. 목록 31.1은 INSERT_SALE의 정의이다.

목록 31.1 INSERT_SALE 저장 프로시저

```
CREATE PROCEDURE INSERT_SALE AS BEGIN EXIT; END ^
...
ALTER PROCEDURE INSERT_SALE (
 ICUSTOMER_ID INTEGER,
 ISALE_DATE DATE,
 ITOTAL_PRICE DOUBLE PRECISION)
RETURNS(
 RSALE_NUMBER INTEGER)
AS
BEGIN
 /* 우선, GEN_SALENO 생성기로부터 새 판매 번호를 얻는다. */
 /* 그리고 그 번호를 리턴값을 뜻하는 rSale 인자에 넣는다. */
 /* 따라서, 이 저장 프로시저를 호출한 클라이언트는 rSale에 담긴 */
 /* 판매 번호를 돌려 받게 된다. */
 rSALE_NUMBER = gen_id(GEN_SALENO, 1);
```

```
/* Now insert the record into the SALES table */
INSERT INTO SALES(
SALE_NUMBER,
CUSTOMER_ID,
SALE_DATE,
TOTAL_PRICE)
VALUES(
:rSALE_NUMBER,
:iCUSTOMER_ID,
:iSALE_DATE,
:iTOTAL_PRICE);
END
```

이 저장 프로시저는 매우 기본적인 SQL 코드를 수행한다. 우선, GEN_SALENO 생성기로부터 새 판매 레코드에 대한 판매 번호를 얻고, 프로시저의 인자들을 간단한 INSERT INTO SQL문에 넣어서 테이블에 새 판매 레코드를 삽입한다.

두 번째 저장 프로시저는 INSERT_SALE_ITEM이라는 것인데, 이 저장 프로시저는 ITEMS 테이블에 개별 판매 항목을 삽입하는 역할을 한다. 이 프로시저는 한 번에 하나의 항목을 삽입하기 때문에 한 고객에게 여러 항목들을 판매한 경우에는 이 프로시저를 여러 번 호출해야 한다. 클라이언트 애플리케이션은 우선 INSERT_SALE 저장 프로시저를 호출해서 새 판매 레코드를 삽입한 후 그 레코드에 대한 판매 번호를 얻고, 각각의 판매 항목에 대해 INSERT_SALE_ITEM을 반복적으로 호출한다. 각각의 호출마다 INSERT_SALE로 얻은 판매 번호를 인자에 넣어야 제대로 된 레코드에 판매 항목 정보가 삽입된다.

INSERT_SALE_ITEM은 판매 번호, 부품 번호, 그리고 그 부품의 판매 수량을 인자로 받아들인다. 이 저장 프로시저는 몇 가지 데이터 무결성 작업을 수행한다. 우선, 이 프로시저는 PART 테이블에 판매할 부품 개수가 충분한지 점검해서 부품 개수가 모자라면 예외를 발생한다. PART 테이블의 해당 부품 개수에서 Qty 인자의 값을 뺀다. 마지막으로 판매 항목을 ITEMS 테이블에 추가한다.

목록 31.2는 INSERT_SALE_ITEM 저장 프로시저의 정의이다.

목록 31.2 INSERT_SALE_ITEM 저장 프로시저

```
CREATE PROCEDURE INSERT_SALE_ITEM AS BEGIN EXIT; END ^
...
ALTER PROCEDURE INSERT_SALE_ITEM (
 ISALE_NUMBER INTEGER,
 IPART_NO VARCHAR(10),
 IQTY SMALLINT)
AS
 DECLARE VARIABLE Actual_Qty VARCHAR(10);
BEGIN
/* PARTS 테이블에 iQTY 개수만큼의 부품이 남아 있는지 확인 */
 SELECT QUANTITY FROM PART
      WHERE PART_NUMBER = :iPART_NO
      INTO Actual_Qty;
 IF (Actual_Qty < iQTY) THEN
      EXCEPTION EXP_EXCESS_ORDER;
 ELSE BEGIN
      /* PART 테이블의 해당 부품 개수를 판매된 개수만큼 감소시킨다. */
      UPDATE PART
      SET QUANTITY = (:Actual_Qty - :iQty)
      WHERE PART_NUMBER = :iPART_NO;
      /* 새 주문을 삽입한다. */
      INSERT INTO ITEMS(
          SALE_NUMBER,
          PART_NO,
          QTY)
      VALUES(
          :iSALE_NUMBER,
          :iPART_NO,
          :iQTY);
 END
 END
```



ISQL 도구를 사용해서 데이터베이스 메타데이터를 입력했다면 종료 문자를 변경해야 한다. 저장 프로시저의 모든 문장들은 세미콜론(;)으로 끝나야 하는데, SQL 질의문 자체의 종료 문자 역시 세미콜론이다. 따라서, SQL 자체의 종료 문자를 세미콜론이 아닌 다른 것으로 변경해야 한다. SQL 종료 문자를 바꾸는 명령은 SET TERM이다. SALES 데이터베이스의 경우, 탈자 기호(^)를 종료 문자로 사용한다. 다음과 같은 SQL 코드를 수행하면 SQL 종료 문자가 ^으로 바뀐다.

```
SET TERM ^;
```

작업 허용 권한 설정

데이터베이스 정의의 마지막 단계는 테이블과 저장 프로시저에 대한 사용자 허용 권한을 설정하는 것이다. 이번 예제 데이터베이스의 경우, CUSTOMER 테이블에 대해서 모든 사용자들에게 SELECT와 UPDATE를 허용한다.

```
GRANT SELECT, UPDATE ON CUSTOMER TO PUBLIC WITH GRANT OPTION;
```

그리고 SALE 테이블에 대해서는 모든 종류의 작업들을 허용한다.

```
GRANT ALL ON SALE TO PUBLIC WITH GRANT OPTION;
```

GRANT OPTION절은 테이블에 접근할 수 있는 사용자에 대해서만 GRANT 다음에 나오는 작업 권한들을 허용한다는 뜻이다. 다음은 이번 예제 애플리케이션의 테이블과 저장 프로시저들에 대한 작업 허용 권한을 설정하는 GRANT문들이다.

```
/* 이 데이터베이스에 대한 허용 설정 */
GRANT SELECT, UPDATE ON CUSTOMER TO PUBLIC WITH GRANT OPTION;
GRANT ALL ON SALES TO PUBLIC WITH GRANT OPTION;
GRANT ALL ON PART TO PUBLIC WITH GRANT OPTION;
GRANT ALL ON ITEMS TO PUBLIC WITH GRANT OPTION;
```

```
GRANT EXECUTE ON PROCEDURE INSERT_SALE TO PUBLIC;
GRANT EXECUTE ON PROCEDURE INSERT_SALE_ITEM TO PUBLIC;
```

이로서 데이터베이스의 정의가 끝났다. 다음은 데이터베이스 객체들의 연결에 관한 설명이다.

데이터베이스 조작의 중앙 집중화 - 비즈니스 규칙

이번에는 사용자 인터페이스로부터 데이터베이스 조작과 비즈니스 규칙(business rules)을 분리시키는 방법을 살펴보자. 데이터베이스 조작과 비즈니스 규칙을 사용자 인터페이스로부터 분리시키는 목적은 여러 가지이다. 우선, 다양한 비즈니스 규칙들이 애플리케이션 전반에 널려 있는 것보다는 비즈니스 규칙을 하나의 데이터 모듈에 몰아 넣는 편이 훨씬 더 관리하기 편하다. 또한, 비즈니스 규칙을 담고 있는 데이터 모듈에 적절한 컴포넌트들을 추가해서 2-티어 구조 모델을 3-티어 구조 모델로 쉽게 전환할 수 있다. 이번 예제에서는 그러한 방법을 사용하지는 않지만, 어쨌든 2-티어 구조 모델을 3-티어 구조 모델로 쉽게 전환할 수 있다는 점을 알아두면 이후에 3-티어 구조 모델의 클라이언트/서버 애플리케이션을 만들 때 도움이 될 것이다.

데이터 조작과 비즈니스 규칙을 맡기기에 딱 좋은 컴포넌트는 바로 TDataModule 컴포넌트이다. 이번 예제에서도 TDataModule을 사용한다.

이번 예제에서는 하나의 TDataModule을 사용한다. 별로 복잡한 애플리케이션이 아니기 때문에 하나로도 충분하다. 크고 복잡한 애플리케이션이라면 업무 규칙들을 기능별/논리별로 단위로 구분해서 여러 개의 TDataModule로 분산시키는 것이 좋을 것이다.

목록 31.3은 TDDGSalesDataModule을 정의하는 SalesDM.pas의 소스 코드이다.

목록 31.3 Ch31\SalesDM.pas

```
unit SalesDM;

interface

uses
```

```
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  DBTables, Db;

type
  TDDGSalesDataModule = class(TDataModule)
    qryCustomer: TQuery;
    dbSales: TDatabase;
    usqlCustomer: TUpdateSQL;
    qryCustomerCUSTOMER_ID: TIntegerField;
    qryCustomerFNAME: TStringField;
    qryCustomerLNAME: TStringField;
    qryCustomerCREDIT_LINE: TSmallintField;
    qryCustomerWORK_ADDRESS: TStringField;
    qryCustomerALT_ADDRESS: TStringField;
    qryCustomerCITY: TStringField;
    qryCustomerSTATE: TStringField;
    qryCustomerZIP: TStringField;
    qryCustomerWORK_PHONE: TStringField;
    qryCustomerALT_PHONE: TStringField;
    qryCustomerCOMMENTS: TMemoField;
    qryCustomerCOMPANY: TStringField;
    qryParts: TQuery;
    usqlParts: TUpdateSQL;
    qryPartsPART_NUMBER: TStringField;
    qryPartsDESCRIPTION: TStringField;
    qryPartsQUANTITY: TSmallintField;
    qryPartsLIST_PRICE: TFloatField;
    qryPartsRETAIL_PRICE: TFloatField;
    qryPartsDEALER_PRICE: TFloatField;
    qryPartsJOBBER_PRICE: TFloatField;
    spInsertSaleItem: TStoredProc;
    spInsertSale: TStoredProc;
    qryTotalPrice: TQuery;
    tblTempItems: TTable;
    tblTempItemsPART_NUMBER: TStringField;
    tblTempItemsDESCRIPTION: TStringField;
    tblTempItemsQUANTITY: TSmallintField;
```

```
  tblTempItemsRETAIL_PRICE: TFloatField;
  tblTempItemsTOTAL_PRICE: TFloatField;
  qryTotalPriceSUMOFTOTAL_PRICE: TFloatField;
  qrySale: TQuery;
  dsCustomer: TDataSource;
  qryItems: TQuery;
  dsSale: TDataSource;
  qrySaleSALE_NUMBER: TIntegerField;
  qrySaleSALE_DATE: TDateTimeField;
  qrySaleTOTAL_PRICE: TFloatField;
  qryItemsDESCRIPTION: TStringField;
  qryItemsQTY: TSmallintField;
  qryCustomerSearch: TQuery;
  procedure tblTempItemsBeforePost(DataSet: TDataSet);
  procedure dbSalesLogin(Database: TDatabase; LoginParams: TStrings);
protected
  procedure SetAfterTempItemsChange(Value: TDataSetNotifyEvent);
  function GetAfterTempItemsChange: TDataSetNotifyEvent;
public

  // 연결 관련 메소드들
  procedure Logout;
  function Login: Boolean;
  function Connect: Boolean;
  procedure Disconnect;

  // 고객 관련 메소드들
  procedure FirstCustomer;
  procedure LastCustomer;
  procedure NextCustomer;
  procedure PrevCustomer;
  procedure EditCustomer;
  procedure NewCustomer;
  procedure AcceptCustomer;
  procedure CancelCustomer;
  procedure DeleteCustomer;
  function IsFirstCustomer: Boolean;
```

```
    function IsLastCustomer: Boolean;
    function GetCustomerName: String;
    function SearchForCustomer: Boolean;
    // 부품 관련 메소드들
    procedure FirstPart;
    procedure LastPart;
    procedure NextPart;
    procedure PrevPart;
    procedure EditPart;
    procedure NewPart;
    procedure AcceptPart;
    procedure CancelPart;
    procedure DeletePart;
    function IsFirstPart: Boolean;
    function IsLastPart: Boolean;
    function SearchForPart: Boolean;

    // 판매 관련 메소드들
    procedure AddItemToSale;
    procedure SaveSale;
    procedure CancelSale;
    function SaleItemsTotalPrice: double;
    procedure OpenTempItems;
    procedure CloseTempItems;

    // 속성들
    property AfterTempItemsChange: TDataSetNotifyEvent
          read GetAfterTempItemsChange
      write SetAfterTempItemsChange;

  end;

var
  DDGSalesDataModule: TDDGSalesDataModule;

implementation
```

```
uses CustomerSrchFrm, LoginFrm;

{$R *.DFM}


procedure TDDGSalesDataModule.SetAfterTempItemsChange(Value:
TDataSetNotifyEvent);
begin
 { 이 쓰기 메소드는 Value 인자를 임시 항목 테이블의 AfterPost 이벤트 속성과
   AfterDelete 이벤트 속성에 설정한다. 데이터가 변경되면 Value로 설정한
   메소드가 수행된다. }
  tblTempItems.AfterPost   := Value;
  tblTempItems.AfterDelete := Value;
end;

function TDDGSalesDataModule.GetAfterTempItemsChange: TDataSetNotifyEvent;
begin
  Result := tblTempItems.AfterPost;
end;

// 연결 관련 메소드들

procedure TDDGSalesDataModule.dbSalesLogin(Database: TDatabase;
  LoginParams: TStrings);
begin
 { GetLoginParams()를 호출해서 Loginparams 문자열 목록에 사용자의 로그인
   정보를 채워 넣는다. GetLoginParams()는 LoginFrm.pas에 정의되어 있다. }
  GetLoginParams(LoginParams);
end;

procedure TDDGSalesDataModule.Logout;
begin
  Disconnect;
end;

function TDDGSalesDataModule.Login: Boolean;
begin
  Result := Connect;
```

```
end;

function  TDDGSalesDataModule.Connect: Boolean;
begin
 { 사용자를 데이터베이스에 연결시킨다. dbSales.Connected를 True로 설정하면
   dbSales의 OnLogin 이벤트가 발생한다. OnLogin 이벤트 처리 메소드는
   LoginFrm.pas에 정의된 커스텀 로그인 대화상자를 띄운다. }
  try
    dbSales.Connected  := True;
    qryCustomer.Active := True;
    qryParts.Active    := True;
    qrySale.Active     := True;
    qryItems.Active    := True;
    Result := True;
  except
    MessageDlg('잘못된 암호이거나 로그인 정보입니다. 로그인할 수 없습니다.',
          mtError, [mbok], 0);
    dbSales.Connected := False;
    Result := False;
  end;
end;

procedure TDDGSalesDataModule.Disconnect;
begin
  // 데이터베이스와의 연결을 끊는다.
  dbSales.Connected := False;
end;

// 고객 관련 메소드들

procedure TDDGSalesDataModule.AcceptCustomer;
begin
  dbSales.ApplyUpdates([qryCustomer]);
end;

procedure TDDGSalesDataModule.CancelCustomer;
begin
  qryCustomer.CancelUpdates;
```

```
end;

procedure TDDGSalesDataModule.DeleteCustomer;
begin
  qryCustomer.Delete;
end;

procedure TDDGSalesDataModule.EditCustomer;
begin
  qryCustomer.Edit;
end;

procedure TDDGSalesDataModule.FirstCustomer;
begin
  qryCustomer.First;
end;

procedure TDDGSalesDataModule.LastCustomer;
begin
  qryCustomer.Last;
end;

procedure TDDGSalesDataModule.NewCustomer;
begin
  qryCustomer.Insert;
end;

procedure TDDGSalesDataModule.NextCustomer;
begin
  qryCustomer.Next;
end;

procedure TDDGSalesDataModule.PrevCustomer;
begin
  qryCustomer.Prior;
end;

function TDDGSalesDataModule.IsFirstCustomer: Boolean;
```

```
begin
  Result := qryCustomer.Bof;
end;

function TDDGSalesDataModule.IsLastCustomer: Boolean;
begin
  Result := qryCustomer.Eof;
end;

function TDDGSalesDataModule.GetCustomerName: String;
begin
 { 보통은 회사 이름을 돌려준다. 회사 이름이 없으면 고객 이름을 돌려준다. }
  if qryCustomerCOMPANY.AsString <> EmptyStr then
    Result := qryCustomerCOMPANY.AsString
  else
    Result := Format('%s %s', [qryCustomerFNAME.AsString,
                               qryCustomerLNAME.AsString]);
end;

function TDDGSalesDataModule.SearchForCustomer: Boolean;
var
  CustID: Integer;
  SearchQry: String;
begin
  // 실패라고 가정한다.
  Result := False;
 { CustomerSrchFrm.pas에 정의되어 있는 SearchCustomer() 함수를 호출한다.
   SearchCustomer() 함수가 돌려준 질의 문자열을 TQuery 컴포넌트인
   qryCustomerSearch에 추가한다. }
  SearchQry := SearchCustomer;
  if SearchQry <> EmptyStr then
  begin
    Screen.Cursor := crSQLWait;
    try
        qryCustomerSearch.Close;
        qryCustomerSearch.SQL.Clear;
        qryCustomerSearch.SQL.Add(SearchQry);
        qryCustomerSearch.Open;
```

```
        try

          // 레코드가 없으면 이 메소드를 종료한다.
          if qryCustomerSearch.FieldByName('CUSTOMER_ID').IsNull then
          begin
            Screen.Cursor := crDefault;
            Exit;
          end;

          { 레코드가 있으면 고객 ID를 얻은 다음, 그 ID를 이용해서 실제 TQuery
            컴포넌트인 qryCustomer에서 레코드를 찾는다. 결과적으로, 커서는
            원하는 고객 정보가 담긴 레코드를 가리키게 된다. }
          CustID := qryCustomerSearch.FieldByName('CUSTOMER_ID').AsInteger;

          { qryCustomer에 레코드가 없으면 데이터베이스의 무결성에 문제가 있는
            것이므로 예외를 발생시킨다. }
          if not qryCustomer.Locate('CUSTOMER_ID', CustID, []) then
            raise Exception.Create('데이터베이스의 비일관성.')
          else
            Result := True;
        finally
          qryCustomerSearch.Close;
        end;
      finally
        Screen.Cursor := crDefault;
      end;
    end
    else
      Result := False;;
end;

// 부품 관련 메소드들

function TDDGSalesDataModule.IsFirstPart: Boolean;
begin
  Result := qryParts.Bof;
end;
```

```
function TDDGSalesDataModule.IsLastPart: Boolean;
begin
  Result := qryParts.Eof;
end;

procedure TDDGSalesDataModule.AcceptPart;
begin
  dbSales.ApplyUpdates([qryParts]);
end;

procedure TDDGSalesDataModule.CancelPart;
begin
  qryParts.CancelUpdates;
end;

procedure TDDGSalesDataModule.DeletePart;
begin
  qryParts.Delete;
end;

procedure TDDGSalesDataModule.EditPart;
begin
  qryParts.Edit;
end;

procedure TDDGSalesDataModule.FirstPart;
begin
  qryParts.First;
end;

procedure TDDGSalesDataModule.LastPart;
begin
  qryParts.Last;
end;

procedure TDDGSalesDataModule.NewPart;
begin
  qryParts.Insert;
```

```
end;

procedure TDDGSalesDataModule.NextPart;
begin
  qryParts.Next;
end;

procedure TDDGSalesDataModule.PrevPart;
begin
  qryParts.Prior;
end;

function TDDGSalesDataModule.SearchForPart: Boolean;
{ 이 메소드는 사용자가 지정한 부품 번호에 해당하는 부품을 찾는다. }
var
  PartNumber: string;
begin
  Result := False;
  PartNumber := '';
  if InputQuery('Part Search', 'Enter a Part Number', PartNumber) then
    if not qryParts.Locate('PART_NUMBER', PartNumber, []) then
      Exit
    else
      Result := True;
end;

// 판매 관련 메소드들

procedure TDDGSalesDataModule.AddItemToSale;
begin
 { tblTempItems는 판매에 추가되는 항목들을 담는 임시 테이블이다.
     나중에 판매를 저장할 때 이 임시 테이블의 레코드들을 이용해서
     실제로 판매를 데이터베이스에 기록하는 저장 프로시저를 호출한다. }
  if not tblTempItems.Locate('PART_NUMBER',
    qryParts.FieldByName('PART_NUMBER').AsString, []) then
  begin
    tblTempItems.Insert;
    try
```

```
      tblTempItems['PART_NUMBER']  := qryParts['PART_NUMBER'];
      tblTempItems['DESCRIPTION']  := qryParts['DESCRIPTION'];
      tblTempItems['QUANTITY']     := 1;
      tblTempItems['RETAIL_PRICE'] := qryParts['RETAIL_PRICE'];
      tblTempItems.Post;
    except
      tblTempItems.Cancel;
    end;
  end
  else
    MessageDlg('아이템이 이미 리스트에 있습니다.', mtWarning, [mbok], 0);
end;

procedure TDDGSalesDataModule.CancelSale;
begin
 { 사용자가 판매를 취소하면 tblTempItems에 추가된 항목들을 지운다. }
  tblTempItems.Close;
  tblTempItems.EmptyTable;
  tblTempItems.Open;
end;

procedure TDDGSalesDataModule.SaveSale;
var
  SaleNo: Integer;
begin
 { 사용자가 판매를 저장하면 이 메소드가 호출된다. 이 메소드는 우선 판매를 만들고,
   키값으로 사용할 판매 번호를 SaleNo 변수에 저장한다. 그런 다음, 임시 테이블인
   tblTempItems에서 SalesNo로 지정된 모든 레코드들을 판매에 추가한다. }
  dbSales.StartTransaction;
  try
    { 우선, 판매 레코드를 만든다. }
    with spInsertSale do
    begin
      ParamByName('iCUSTOMER_ID').AsInteger := qryCustomer['CUSTOMER_ID'];
      ParamByName('iSALE_DATE').AsDateTime  := Now;
      ParamByName('iTOTAL_PRICE').AsFloat   := SaleItemsTotalPrice;
      ExecProc;
      // SaleNo에서 키값을 얻는다.
```

```
      SaleNo := ParamByName('rSALE_NUMBER').AsInteger;
    end;

    // 임시 테이블 tblTempItems에서 SalesNo로 지정된 모든 레코드들을
    // 판매에 추가한다.
    tblTempItems.First;
    while not tblTempItems.Eof do
    begin
      with spInsertSaleItem do
      begin
        ParamByName('IPART_NO').AsString :=  tblTempItems['PART_NUMBER'];
        ParamByName('IQTY').AsInteger := tblTempItems['QUANTITY'];
        ParamByName('ISALE_NUMBER').AsInteger := SaleNo;
        ExecProc;
      end;
      tblTempItems.Next;
    end;

    dbSales.Commit;

    // 수정된 테이블들을 최신 정보로 고친다.
    qryParts.Close;
    qryParts.Open;

    tblTempItems.Close;
    tblTempItems.EmptyTable;
    tblTempItems.Open;

  except
    dbSales.Rollback;
  end;
end;

function TDDGSalesDataModule.SaleItemsTotalPrice: double;
begin
 { qryTotalPrice를 이용해서 tblTempItmes 테이블에 추가된 모든 레코드들의
   판매액 총합을 얻는다. 이 데이터 모듈을 사용하는 모든 폼은 이 메소드를 호출할
   수 있다. }
```

```
  qryTotalPrice.Close;
  qryTotalPrice.Open;
  try
    Result := qryTotalPrice.FieldByName('SUM OF TOTAL_PRICE').AsFloat;
  finally
    qryTotalPrice.Close;
  end;
end;

procedure TDDGSalesDataModule.tblTempItemsBeforePost(DataSet: TDataSet);
begin
 { 레코드를 임시 테이블에 보내기(post) 전에 사용자가 추가한 항목의 개수에
   기반해서 총 판매액을 구한 후 TOTAL_PRICE 필드에 넣는다. }
  tblTempItemsTOTAL_PRICE.ReadOnly := False;
  try
    tblTempItems['TOTAL_PRICE'] := tblTempItems['RETAIL_PRICE'] *
      tblTempItems['QUANTITY'];
  finally
    tblTempItemsTOTAL_PRICE.ReadOnly := True;
  end;
end;

procedure TDDGSalesDataModule.OpenTempItems;
begin
  tblTempItems.Close;
  tblTempItems.EmptyTable;
  tblTempItems.Open;
end;

procedure TDDGSalesDataModule.CloseTempItems;
begin
  tblTempItems.Active := False;
end;

 end.
```

TDDGSalesDataModule에는 TDatabase 컴포넌트인 dbSales와 여러 가지 TQuery, TUpdateSQL, TStoredProc 컴포넌트들이 포함되어 있다.

dbSales는 SALES.GDB에 존재하는 SQL 백 엔드와의 연결을 담당한다. dbSales는 DBExplorer 프로그램으로 설정된 DDGSALES 대표명(alias)을 이용해서 애플리케이션 수준의 대표명인 DDGSalesDB를 설정한다. 애플리케이션이 시작되는 초기에는 dbSales의 Connected 속성이 False이므로 dbSales에 속한 모든 테이블들은 열리지 않은 상태이다. dbSales에는 OnLogin 이벤트 처리 메소드가 있는데, 이에 대해서는 잠시 후에 다시 이야기하겠다.

목록 31.3을 보면 TDDGSalesDataModule의 메소드들이 기능별로 분류되어 있음을 알 수 있다. 각각의 메소드들은 다음과 같이 분류되어 있다.

- 연결 관련 메소드들 : 사용자를 데이터베이스에 로그인/로그오프시키는 데 관련된 메소드들
- 고객 관련 메소드들 ; 고객 데이터(CUSTOMER 테이블)를 조작하는 메소드들
- 부품 관련 메소드들 : 부품 데이터(PART 테이블)를 조작하는 메소드들
- 판매 관련 메소드들 : 판매 데이터(SALES 테이블)를 만들고 관리하는 메소드들

각각의 메소드들은 코드와 주석만으로도 쉽게 이해할 수 있을 것이다. 그 중 SaveSale()이란 메소드는 TStoredProc 컴포넌트를 이용해서 새 판매를 만들고, 판매 항목들을 판매에 추가한다. 그 메소드가 사용하는 저장 프로시저들은 목록 31.1과 31.2에 나와 있다.

Login()/Logout() 메소드

Login()과 Logout()은 각각 사용자의 로그인과 로그아웃을 담당하는 메소드이다. Login()은 Connect() 메소드를 호출한다. 그러면 dbSales를 거쳐서 데이터베이스와의 연결이 설정된다. Connect()는 dbSales.Connected 속성을 True로 하며, 그러면 dbSales의 OnLogin 이벤트가 발생한다. 이 이벤트에 연결된 메소드는 dbSalesLogin()이다. 이 메소드는 LoginFrm.pas에 정의된 GetLoginParams()를 호출해서 로그인 대화상자에 입력된 사용자 로그인 정보를 얻는다. 목록 31.4에 GetLoginParams() 메소드가 나와 있다.

목록 31.4 Ch31\LoginFrm.pas

```
unit LoginFrm;

interface

uses WinTypes, WinProcs, Classes, Graphics, Forms, Controls, StdCtrls,
  Buttons, ExtCtrls;

type
  TLoginForm = class(TForm)
    lblEnterPassword: TLabel;
    lblEnterName: TLabel;
    edtName: TEdit;
    edtPassword: TEdit;
    btnOK: TButton;
    btnCancel: TButton;
  public
  end;

function GetLoginParams(ALoginParams: TStrings): Boolean;

implementation

{$R *.DFM}

function GetLoginParams(ALoginParams: TStrings): Boolean;
var
  LoginForm: TLoginForm;
begin
  Result := False;
  LoginForm := TLoginForm.Create(Application);
  try
    if LoginForm.ShowModal = mrOk then
    begin
      ALoginParams.Values['USER NAME'] := LoginForm.edtName.Text;
      ALoginParams.Values['PASSWORD'] := LoginForm.edtPassWord.Text;
```

```
      Result := True;
    end;
  finally
    LoginForm.Free;
  end;
end;

end.
```

Logout() 메소드는 그냥 dbSales를 닫는 역할을 한다. 결과적으로, 모든 TQuery/TTable 컴포넌트들의 연결도 닫힌다.

CUSTOMER 테이블 조작 메소드들

DDGSalesDataModule의 NewCustomer(), AcceptCustomer, EditCustmoer(), Delete Customer(), CancelCustomer()는 CUSTOMER 테이블을 조작하는 메소드들이다. 이 메소드들은 모두 TQuery의 해당 메소드를 호출해서 자신의 작업을 수행한다. 그 외에도 몇 가지 고객 관련 메소드들이 있는데, 이들에 대해서는 좀더 설명이 필요할 것 같다.

GetCustomerName()은 고객의 회사 이름을 돌려주는 메소드이다. 회사 이름이 존재하지 않으면 CustID 인자로 지정된 고객 ID에 해당하는 고객의 이름과 성을 돌려준다.

SearchForCustomer()는 CUSTOMER 테이블에서 특정 고객을 검색하는 메소드이다. 이 메소드는 사용자가 고객 검색 폼에서 지정한 값들을 질의문을 만든 후 질의를 수행해서 고객을 검색한다. 질의문을 만들어내는 함수는 고객 검색 폼에 정의되어 있는데, 고객 검색 폼에 대해서는 잠시 후에 설명하겠다. 어쨌든, 이 메소드는 만들어진 질의문을 qryCustomerSearch.SQL에 넣는다고만 알아두면 된다. 지정된 고객이 존재하는 경우, 현재 레코드는 그 고객을 가리키게 된다.

PART 테이블 조작 메소드들

PART 테이블 조작 메소드들도 CUSTOMER 테이블 조작 메소드들과 매우 비슷하다. NewPart(), EditPart(), AcceptPart(), DeletePart(), CancelPart()는 각각 TQuery의 해당 메소드를 호출하는 간단한 메소드들이다.

SearchForPart()는 SearchForCustomer()에 비해 간단한 편이다.

SearchForPart()는 InputQuery() 함수를 이용해서 부품 번호를 얻은 다음, Locate()를 이용해서 부품 번호에 해당하는 부품을 찾는다.

SALES 테이블 조작 메소드

SALES 테이블을 조작하는 메소드들은 클라이언트/서버 데이터베이스 애플리케이션에 관련된 다양한 기법을 보여준다. 특히, SaveSale() 메소드가 주목할 만하다.

우선, AddItemSale()은 사용자에게 새 판매에 추가할 항목들을 지정할 수 있게 하기 위한 것이다(그림 31.2).

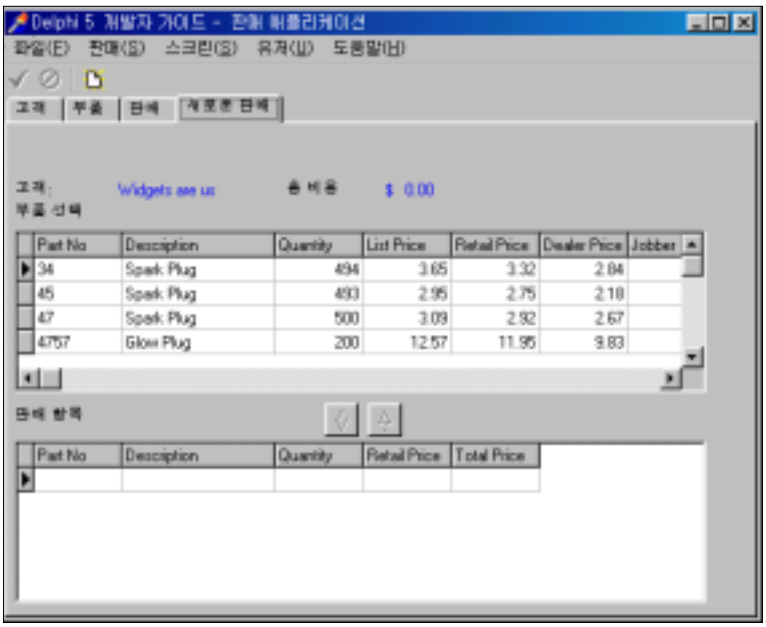


그림 31.2
판매에 항목들을 추가

CancelSale()은 판매 추가 작업을 중지한다.

SaveSale()은 DDGSalesDataModule에서 가장 복잡한 메소드이다. 이 메소드는 dbSales의 트랜잭션 기능을 이용해서 판매를 데이터베이스에 추가한다. 이 메소드는 트랜잭션을 시작하고, 판매 레코드를 추가하며, 판매할 임의의 개수의 항목들을 추가하고, 예외 발생 여부에 따라서 전체 트랜잭션을 완료(commit)하거나 취소(rollback)시킨다.

판매 레코드를 추가하는 것은 spInsertSale이라는 TStoredProc 컴포넌트로 지정된 저장 프로시저가 담당한다. 저장 프로시저가 생성한 판매 번호를 얻는 부분을 주목하기 바란다.

```
SaleNo := ParamByName('rSALE_NUMBER').AsInteger;
```

TStoredProc 컴포넌트 spInsertSaleItems가 각 레코드들을 ITEMS 테이블에 추가할 때 SaleNo를 사용해서 적절한 판매 레코드를 참조한다. 이렇게 하면 판매된 부품들과 재고 사이의 데이터 무결성이 보장된다.

임시 테이블 조작 메소드들

나머지 메소드들은 판매할 항목들을 임시로 저장하는 데 쓰이는 임시 테이블(파라독스 테이블이다)을 조작하는 메소드들이다. 재정의는 임시 테이블 TEMPITEMS.DB의 정의이다.



TEMPITEMS.DB 테이블의 필드들

필드 이름	형	크기	의미
PART_NO	A	10	이 항목의 부품 번호
DESCRIPTION	A	18	이 항목에 대한 설명
QUANTITY	S		판매될 부품의 개수
RETAIL_PRICE	N	50	판매될 부품의 단가
TOTAL_PRICE	N	50	판매될 부품들의 가격(단가×개수)

AddItemToSale() 메소드는 판매에 항목(부품)들을 추가하는 작업을 담당한다.

SaleItemsTotalPrice() 메소드는 tblTempItems에 존재하는 모든 항목들의 가격 총합을 돌려준다. 이 메소드는 qryTotalPrice 컴포넌트를 이용해서 가격 총합을 얻기 위한 SQL 질의문을 수행한다. SQL 질의문은 다음과 같다.

```
select SUM(TOTAL_PRICE) from tempitems.db
```

이 문장은 TOTAL_PRICE 필드의 숫자값들을 모두 더한 값을 돌려준다.

tblTempItemsBeforePost() 메소드는 tblTempItems.BeforePost 이벤트의 처리 메소드이다. TOTAL_PRICE 필드의 값은 사용자가 직접 입력하지 않는다. 판매 레코드를 테이블에 보내기(post) 전에 이 필드에 판매할 부품의 단가와 개수를 곱한 값을 넣어야 하는데, Post가 일어나기 전에 BeforePost 이벤트가 발생하므로 BeforePost 이벤트 처리 메소드에서 TOTAL_PRICE 필드의 값을 설정하면 된다.

데이터 조작 컴포넌트의 이벤트들을 TDataModule의 사용자에게 전달하려면

사용자 인터페이스 요소들은 데이터 조작 컴포넌트들의 이벤트에 의존하는 경우가 많다. 그런데 TDataModule 안에 포함된 데이터 조작 컴포넌트들의 이벤트들이 자동적으로 사용자 인터페이스 컴포넌트들에게 전달되지는 않는다는 점이 문제이다. 데이터 조작을 중앙 집중화하려면 데이터 조작 컴포넌트들을 TDataModule 안에 집어넣는 것을 피할 수가 없으므로 이벤트 전달 문제를 해결할 수 있어야 제대로 된 클라이언트/서버 애플리케이션이 될 수 있다.

한 가지 방법은 TDataModule에 새로운 이벤트를 추가하고, 그 이벤트를 통해서 TDataModule 안에 있는 데이터 조작 컴포넌트들의 이벤트를 사용자에게 전달하는 것이다. 이번 예제의 경우, tblTempItems의 AfterPost와 AfterDelete 이벤트가 TDataModule의 AfterTempItemsChange 이벤트 속성을 통해서 사용자에게 전달된다. AfterTemp ItemsChange 이벤트 속성의 읽기/쓰기 메소드를 통해서 tblTempItems의 두 이벤트 속성을 조작하는 부분을 참고하기 바란다.

사용자 인터페이스의 설계

데이터 조작의 중앙 집중화를 구현해 내었으므로 이제는 TDataModule의 메소드, 속성, 이벤트들에 관련된 사용자 인터페이스를 설계할 차례이다. 그럼 이번 예제 애플리케이션이 사용하는 여러 폼들에 대해서 알아보자.

이번 예제 애플리케이션은 하나의 폼이 일반 윈도우로도, 자식 윈도우로도 될 수 있는 프레임웍(4장에 나왔던 것)을 사용한다.

이번 예제 애플리케이션의 폼 구조는 그림 31.3과 같다.

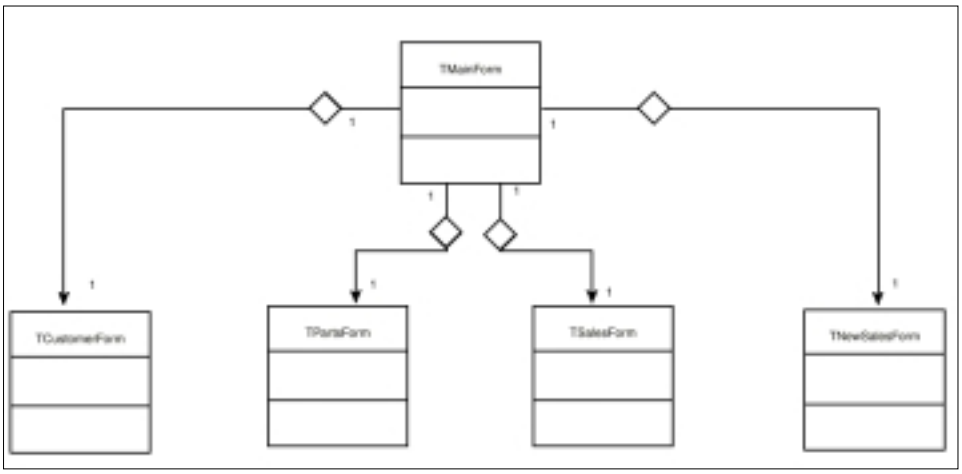


그림 31.3
재고 관리 응용 프로그램의 폼 구조

메인 폼에 담을 수 있는 자식 폼의 종류는 다음과 같다.

- 고객폼 : 고객을 추가, 수정, 조회하는 데 쓰인다.
- 부품폼 : 부품 재고를 추가, 수정, 조회하는 데 쓰인다.
- 판매폼 : 판매를 조회하는 데 쓰인다.
- 새로운 판매폼 : 새 판매를 추가하는 데 쓰인다.

메인 폼의 자식 폼으로 쓰이지 않는 다른 폼들도 몇 가지 있는데, 그것들에 대해서는 나중에 다시 설명하겠다. 그럼 메인 폼과 각각의 자식 폼들에 대해서 알아보자.

TMainForm - 애플리케이션의 메인 폼

애플리케이션의 메인 폼에는 자식 폼들의 부모 컴포넌트 역할을 하는 TTabControl이 있다. 사용자는 메인 폼의 메인 메뉴나 TTabControl 컴포넌트인 tcMain의 탭을 통해서 각각의 자식 폼을 선택한다. 메뉴 항목들과 탭들의 작동은 적절한 이벤트 처리 메소드들을 통해서 동기화된다. 메인 폼의 주된 기능은 한 번에 단 하나의 자식 폼만 생성/활성화되도록 그리고 나머지 자식 폼들은 적절히 해제되도록 관리하는 것이다.

목록 31.5는 메인 폼인 TMainform의 소스 코드이다.

목록 31.5 Ch31\MainFrm.pas

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  Menus, StdCtrls, ComCtrls, ExtCtrls, ChildFrm;

type

 { 메인 폼이 표시하는 자식 폼의 종류는 네 가지이다. TActiveScreenType은 현재
   활성화된 자식 폼의 종류를 판단할 때 사용할 열거형이다. }

  TActiveScreenType = (acCustomer, acParts, acSales, acNewSales);

  TMainForm = class(TForm)
    mmSales: TMainMenu;
    mmiScreen: TMenuItem;
    mmiCustomer: TMenuItem;
    mmiParts: TMenuItem;
    mmiNewSale: TMenuItem;
    mmiSales: TMenuItem;
    mmiFile: TMenuItem;
    mmiExit: TMenuItem;
    mmiHelp: TMenuItem;
    tcMain: TTabControl;
    mmiUser: TMenuItem;
```

```
    mmiLogon: TMenuItem;
    mmiLogoff: TMenuItem;
    imgCar: TImage;
    procedure ScreenClick(Sender: TObject);
    procedure mmiExitClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure tcMainChange(Sender: TObject);
    procedure tcMainChanging(Sender: TObject; var AllowChange: Boolean);
    procedure mmiLogonClick(Sender: TObject);
    procedure mmiLogoffClick(Sender: TObject);
  private
    // ActiveScreenType에는 현재 활성화된 자식 폼의 종류가 저장된다.
    ActiveScreenType: TActiveScreenType;
    // ActiveScreen은 활성화된 자식 폼에 대한 참조이다.
    ActiveScreen: TChildForm;
    procedure SetActiveScreen;
  public
    { Public declarations }
  end;

var
  MainForm: TMainForm;

implementation

uses CustomerFrm, PartsFrm, NewSalesFrm, SalesFrm, SalesDM;

{$R *.DFM}

procedure TMainForm.FormCreate(Sender: TObject);
begin
  // 메인 탭 컨트롤을 메인 폼의 클라이언트 영역에 꽉 차게 한다.
  tcMain.Align := alClient;
end;

procedure TMainForm.mmiExitClick(Sender: TObject);
begin
  Close;
```

```
end;

procedure TMainForm.ScreenClick(Sender: TObject);
begin
{ 사용자가 자식 폼을 변경하기 위해 메인 메뉴 항목을 선택하면 이 메소드가
  수행된다. 이 메소드는 우선 활성화된 자식 폼을 변경할 수 있는지 점검한다. 현재
  활성화된 자식 폼의 CanChange() 메소드가 True를 돌려주면 변경할 수 있다는
  것이므로 전역 변수 ActiveScreenType의 값을 갱신하고, 다른 자식 폼을 실제로
  활성화시키는 메소드인 SetActiveScreen()을 호출한다. }
  if Sender is TMenuItem then
  begin
    if ActiveScreen <> nil then
    begin
      if ActiveScreen.CanChange then
      begin

        TMenuItem(Sender).Checked := True;
        if Sender = mmiCustomer then
          ActiveScreenType := acCustomer
        else if Sender = mmiParts then
          ActiveScreenType := acParts
        else if Sender = mmiSales then
          ActiveScreenType := acSales
        else if Sender = mmiNewSale then
          ActiveScreenType := acNewSales;

        // 선택된 메뉴 항목에 맞게 TTabControl의 현재 탭을 갱신한다.
        tcMain.TabIndex := ord(ActiveScreenType);
        SetActiveScreen;
      end
    end;
  end;
end;

procedure TMainForm.tcMainChange(Sender: TObject);
begin
{ 이 메소드는 사용자가 선택한 탭에 해당하는 자식 폼을 활성화시킨다. 우선,
  선택된 탭에 해당하는 메인 메뉴 항목에 체크 표시를 한 다음, 자식 폼을 실제로
```

```
  활성화시키는 메소드인 SetActiveScreen()을 호출한다. }
  if ActiveScreen <> nil then
  begin
    case tcMain.TabIndex of
      0: mmiCustomer.Checked := True;
      1: mmiParts.Checked    := True;
      2: mmiSales.Checked    := True;
      3: mmiNewSale.Checked  := True;
    end;
    ActiveScreenType := TActiveScreenType(tcMain.TabIndex);
    SetActiveScreen;
  end;
end;

procedure TMainForm.SetActiveScreen;
{ 이 메소드는 새로운 자식 폼을 활성화시킨다. 자식 폼의 부모는 TTabControl
  컴포넌트인 tcMain이다. }
var
  i: integer;
  TempScreen: TChildForm;
begin
 { 자식 폼의 인스턴스가 생성되어 있으면 자식 폼의 메뉴를 분리시킨 다음, 자식 폼
   인스턴스를 해제한다. }

  TempScreen := ActiveScreen;

  // 메뉴를 분리시킨다.
  if Assigned(ActiveScreen) then
    if ActiveScreen.GetFormMenu <> nil then
      mmSales.UnMerge(ActiveScreen.GetFormMenu);

 { 선택된 자식 폼의 종류에 따라 적절한 자식 폼 인스턴스를 생성한 다음,
   자식 폼의 툴바를 부모 컴포넌트에 결합시킨다. }
  case ActiveScreenType of
    acCustomer:
      begin
        ActiveScreen := TCustomerForm.Create(Application, tcMain);
        TCustomerForm(ActiveScreen).SetToolBarParent(self);
```

```
      end;
    acParts:
      begin
        ActiveScreen := TPartsForm.Create(Application, tcMain);
        TPartsForm(ActiveScreen).SetToolBarParent(self);
      end;
    acSales:
        ActiveScreen := TSalesForm.Create(Application, tcMain);
    acNewSales:
      begin
        ActiveScreen := TNewSalesForm.Create(Application, tcMain);
        TNewSalesForm(ActiveScreen).SetToolBarParent(self);
      end;
  end;
  // 자식 폼의 메뉴를 메인 폼의 메뉴에 결합한다.
  if ActiveScreen <> nil then
  begin
    if ActiveScreen.GetFormMenu <> nil then
      mmSales.Merge(ActiveScreen.GetFormMenu);
    ActiveScreen.Show;
  end;

  if Assigned(TempScreen) then
    TempScreen.Free;

end;

procedure TMainForm.tcMainChanging(Sender: TObject;
  var AllowChange: Boolean);
begin
  // 현재 자식 폼을 변경할 수 있는 상황일 때에만 자식 폼의 변경을 허용한다.
  AllowChange := ActiveScreen.CanChange;
end;

procedure TMainForm.mmiLogonClick(Sender: TObject);
begin
  // 사용자를 시스템에 로그인시킨다.
  if DDGSalesDataModule.Login then
```

```
  begin
    tcMain.Align := alClient;
    tcMain.Visible := True;
    ActiveScreenType := acCustomer;
    SetActiveScreen;
    mmiScreen.Enabled := True;
    mmiLogon.Enabled  := False;
    mmiLogoff.Enabled := True;
  end;
end;

procedure TMainForm.mmiLogoffClick(Sender: TObject);
begin
  // 사용자를 시스템으로부터 로그오프시킨다.
  if Assigned(ActiveScreen) then
  begin
    if ActiveScreen.GetFormMenu <> nil then
      mmSales.UnMerge(ActiveScreen.GetFormMenu);
    ActiveScreen.Free;
    ActiveScreen := nil;
  end;

  tcMain.Visible := False;
  DDGSalesDataModule.Logout;

  mmiScreen.Enabled := False;
  mmiLogon.Enabled  := True;
  mmiLogoff.Enabled := False;

end;

 end.
```

메인 폼은 코드와 주석만으로도 쉽게 이해할 수 있을 것이다. 그럼 실제적인 사용자 인터페이스 기능을 가지고 있는 각각의 자식 폼들에 대해서 알아보자.

TCustomerForm - 고객 입력 폼

TCustomerForm은 데이터베이스에서 고객을 추가, 수정, 삭제하기 위한 자식 폼이다. 이 폼의 모습은 그림 31.4에 나와 있다. 작업에 필요한 대부분의 기능은 이 자식 폼의 조상 폼인 TDBNavStatForm에 들어 있기 때문에 이 폼의 코드는 별로 많지 않다. 목록 31.6은 TCustomerForm의 소스 코드이다.

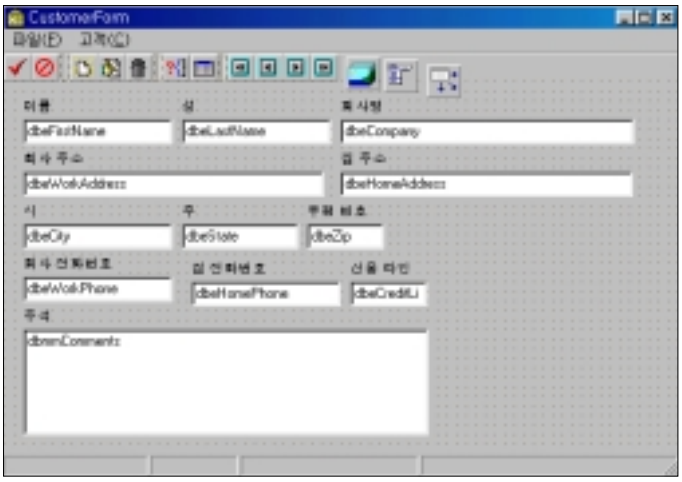


그림 31.4
TCustomerForm - 고객 입력 폼

목록 31.6 Ch31\CustmorFrm.pas

```
unit CustomerFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, DBNavStatFrm, StdCtrls, DBCtrls, Mask, Menus, ImgList,
  ComCtrls, ToolWin, Db, DBModeFrm;

type
  TCustomerForm = class(TDBNavStatForm)
    lblFirstName: TLabel;
    dbeFirstName: TDBEdit;
```

```
    lblLastName: TLabel;
    dbeLastName: TDBEdit;
    lblCreditLine: TLabel;
    dbeCreditLine: TDBEdit;
    lblWorkAddress: TLabel;
    dbeWorkAddress: TDBEdit;
    lblHomeAddress: TLabel;
    dbeHomeAddress: TDBEdit;
    lblCity: TLabel;
    dbeCity: TDBEdit;
    lblState: TLabel;
    dbeState: TDBEdit;
    lblZipCode: TLabel;
    dbeZip: TDBEdit;
    lblWorkPhone: TLabel;
    dbeWorkPhone: TDBEdit;
    lblHomePhone: TLabel;
    dbeHomePhone: TDBEdit;
    lblComments: TLabel;
    dbmmComments: TDBMemo;
    lblCompany: TLabel;
    dbeCompany: TDBEdit;
    dsCustomer: TDataSource;
    EXit1: TMenuItem;
    procedure sbFirstClick(Sender: TObject);
    procedure sbPrevClick(Sender: TObject);
    procedure sbNextClick(Sender: TObject);
    procedure sbLastClick(Sender: TObject);
    procedure sbInsertClick(Sender: TObject);
    procedure sbEditClick(Sender: TObject);
    procedure sbDeleteClick(Sender: TObject);
    procedure sbCancelClick(Sender: TObject);
    procedure sbAcceptClick(Sender: TObject);
    procedure FormShow(Sender: TObject);
    procedure sbFindClick(Sender: TObject);
    procedure sbBrowseClick(Sender: TObject);
  private
    procedure SetNavButtons;
```

```
  public
    function GetFormMenu: TMainMenu; override;
    function CanChange: Boolean; override;
  end;

var
  CustomerForm: TCustomerForm;

implementation

uses SalesDM;

{$R *.DFM}

procedure TCustomerForm.SetNavButtons;
begin
  // 폼의 상태에 맞게 레코드 이동 버튼들을 적절히 활성화/비활성화시킨다.
  sbFirst.Enabled := not DDGSalesDataModule.IsFirstCustomer;
  sbLast.Enabled  := not DDGSalesDataModule.IsLastCustomer;
  sbPrev.Enabled  := not DDGSalesDataModule.IsFirstCustomer;
  sbNext.Enabled  := not DDGSalesDataModule.IsLastCustomer;

  // 레코드 이동 버튼들에 맞게 메뉴 항목들도 적절히 활성화/비활성화시킨다.
  mmiFirst.Enabled    := sbFirst.Enabled;
  mmiLast.Enabled     := sbLast.Enabled;
  mmiPrevious.Enabled := sbPrev.Enabled;
  mmiNext.Enabled     := sbNext.Enabled;
end;

procedure TCustomerForm.sbFirstClick(Sender: TObject);
begin
  // 결과셋의 처음 레코드로 이동한다.
  inherited;
  DDGSalesDataModule.FirstCustomer;
  SetNavButtons;
end;

procedure TCustomerForm.sbPrevClick(Sender: TObject);
begin
  // 결과셋의 이전 레코드로 이동한다.
```

```
  inherited;
  DDGSalesDataModule.PrevCustomer;
  SetNavButtons;
end;

procedure TCustomerForm.sbNextClick(Sender: TObject);
begin
  // 결과셋의 다음 레코드로 이동한다.
  inherited;
  DDGSalesDataModule.NextCustomer;
  SetNavButtons;
end;

procedure TCustomerForm.sbLastClick(Sender: TObject);
begin
  // 결과셋의 마지막 레코드로 이동한다.
  inherited;
  DDGSalesDataModule.LastCustomer;
  SetNavButtons;
end;

procedure TCustomerForm.sbInsertClick(Sender: TObject);
begin
  // 새 고객을 삽입한다.
  inherited;
  DDGSalesDataModule.NewCustomer;
end;

procedure TCustomerForm.sbEditClick(Sender: TObject);
begin
  // 현재 고객을 수정한다.
  inherited;
  DDGSalesDataModule.EditCustomer;
end;

procedure TCustomerForm.sbDeleteClick(Sender: TObject);
begin
  // 현재 고객을 삭제한다.
  inherited;
  DDGSalesDataModule.DeleteCustomer;
```

```
end;

procedure TCustomerForm.sbCancelClick(Sender: TObject);
begin
  // 수정이나 삽입 작업을 취소한다.
  inherited;
  DDGSalesDataModule.CancelCustomer;
end;

procedure TCustomerForm.sbAcceptClick(Sender: TObject);
begin
  // 수정이나 삽입 작업을 확정한다.
  inherited;
  DDGSalesDataModule.AcceptCustomer;
end;

procedure TCustomerForm.FormShow(Sender: TObject);
begin
  // 메뉴 항목들과 버튼들을 적절히 초기화한다.
  inherited;
  SetNavButtons;
end;

function TCustomerForm.CanChange: Boolean;
begin
  // 레코드를 조회하는 상태에서만 자식 폼의 변경이 가능하다.
  Result := FormMode = fmBrowse;
end;

function TCustomerForm.GetFormMenu: TMainMenu;
begin
 { 자식 폼의 메인 메뉴를 돌려준다. 메인 폼은 이 메소드가 돌려준 메뉴를 자신의
   메인 메뉴에 합친다. }
  Result := mmFormMenu;
end;

procedure TCustomerForm.sbFindClick(Sender: TObject);
begin
 // 고객 검색 폼을 띄워서 고객을 검색하도록 한다.
  inherited;
```

```
  DDGSalesDataModule.SearchForCustomer;
end;

procedure TCustomerForm.sbBrowseClick(Sender: TObject);
begin
 // 폼을 조회(browse) 모드로 설정한다. 수정이나 삽입 작업은 취소시킨다.
  inherited;
  if not (FormMode = fmBrowse) then
    DDGSalesDataModule.CancelCustomer;
end;

 end.
```

각 메소드의 기능이나 역할에 대해서는 주석을 참고하기 바란다. 데이터베이스 관련 기능은 대부분 TDDGSalesDataModule에 들어 있기 때문에 코드가 별로 길지 않다. 나머지 자식 폼들도 이 폼과 거의 비슷하다.

TPartsForm - 재고 입력 폼

그림 31.5는 재고 입력 폼인 TPartsForm의 모습이다. 이 폼의 소스 코드는 목록 31.7에 나와 있다.

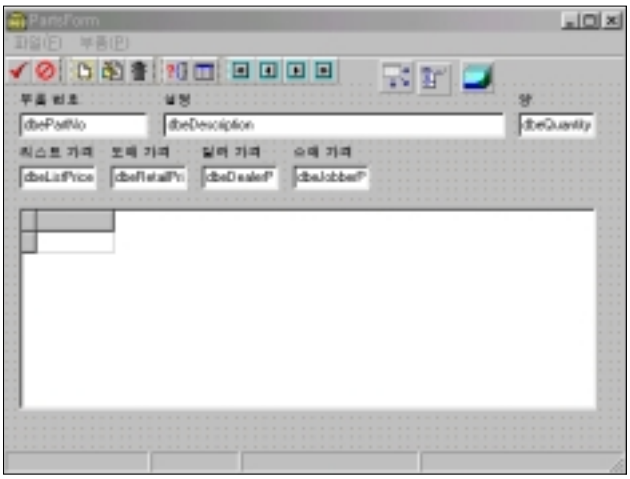


그림 31.5
TPartsForm - 재고 데이터 입력 폼

목록 31.7 Ch31\PartsFrm.pas

```
unit PartsFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  DBNAVSTATFRM, Menus, ImgList, ComCtrls, ToolWin, Grids, DBGrids, Db,
  StdCtrls, Mask, DBCtrls, DBModeFrm;

type
  TPartsForm = class(TDBNavStatForm)
    lblPartNo: TLabel;
    dbePartNo: TDBEdit;
    dsParts: TDataSource;
    lblDescription: TLabel;
    dbeDescription: TDBEdit;
    lblQuantity: TLabel;
    dbeQuantity: TDBEdit;
    lblListPrice: TLabel;
    dbeListPrice: TDBEdit;
    lblRetailPrice: TLabel;
    dbeRetailPrice: TDBEdit;
    lblDealerPrice: TLabel;
    dbeDealerPrice: TDBEdit;
    lblJobberPrice: TLabel;
    dbeJobberPrice: TDBEdit;
    dbgParts: TDBGrid;
    procedure sbAcceptClick(Sender: TObject);
    procedure sbCancelClick(Sender: TObject);
    procedure sbInsertClick(Sender: TObject);
    procedure sbEditClick(Sender: TObject);
    procedure sbDeleteClick(Sender: TObject);
    procedure sbFirstClick(Sender: TObject);
    procedure sbPrevClick(Sender: TObject);
```

```
    procedure sbNextClick(Sender: TObject);
    procedure sbLastClick(Sender: TObject);
    procedure FormShow(Sender: TObject);
    procedure sbFindClick(Sender: TObject);
    procedure sbBrowseClick(Sender: TObject);
  private
    procedure SetNavButtons;
  public
    function GetFormMenu: TMainMenu; override;
    function CanChange: Boolean; override;
  end;

var
  PartsForm: TPartsForm;

implementation

uses SalesDM;

{$R *.DFM}

procedure TPartsForm.SetNavButtons;
begin
  // 폼의 상태에 맞게 레코드 이동 버튼들을 적절히 활성화/비활성화시킨다.
  sbFirst.Enabled := not DDGSalesDataModule.IsFirstPart;
  sbLast.Enabled  := not DDGSalesDataModule.IsLastPart;
  sbPrev.Enabled  := not DDGSalesDataModule.IsFirstPart;
  sbNext.Enabled  := not DDGSalesDataModule.IsLastPart;

  // 레코드 이동 버튼들에 맞게 메뉴 항목들도 적절히 활성화/비활성화시킨다.
  mmiFirst.Enabled := sbFirst.Enabled;
  mmiLast.Enabled  := sbLast.Enabled;
  mmiPrevious.Enabled  := sbPrev.Enabled;
  mmiNext.Enabled  := sbNext.Enabled;

end;

procedure TPartsForm.sbAcceptClick(Sender: TObject);
```

```
begin
  // 수정이나 삽입 작업을 확정한다.
  inherited;
  DDGSalesDataModule.AcceptPart;
end;

procedure TPartsForm.sbCancelClick(Sender: TObject);
begin
  // 수정이나 삽입 작업을 취소한다.
  inherited;
  DDGSalesDataModule.CancelPart;
end;

procedure TPartsForm.sbInsertClick(Sender: TObject);
begin
  // 새 부품을 삽입한다.
  inherited;
  DDGSalesDataModule.NewPart;
end;

procedure TPartsForm.sbEditClick(Sender: TObject);
begin
  // 현재 부품을 수정한다.
  inherited;
  DDGSalesDataModule.EditPart;
end;

procedure TPartsForm.sbDeleteClick(Sender: TObject);
begin
  // 현재 부품을 삭제한다.
  inherited;
  DDGSalesDataModule.DeletePart;
end;

procedure TPartsForm.sbFirstClick(Sender: TObject);
begin
  // 결과셋의 처음 레코드로 이동한다.
  inherited;
```

```
  DDGSalesDataModule.FirstPart;
  SetNavButtons;
end;

procedure TPartsForm.sbPrevClick(Sender: TObject);
begin
  // 결과셋의 이전 레코드로 이동한다.
  inherited;
  DDGSalesDataModule.PrevPart;
  SetNavButtons;
end;

procedure TPartsForm.sbNextClick(Sender: TObject);
begin
  // 결과셋의 다음 레코드로 이동한다.
  inherited;
  DDGSalesDataModule.NextPart;
  SetNavButtons;
end;

procedure TPartsForm.sbLastClick(Sender: TObject);
begin
  // 결과셋의 마지막 레코드로 이동한다.
  inherited;
  DDGSalesDataModule.LastPart;
  SetNavButtons;
end;

procedure TPartsForm.FormShow(Sender: TObject);
begin
  // 메뉴 항목들과 버튼들을 적절히 초기화한다.
  inherited;
  SetNavButtons;
end;

function TPartsForm.CanChange: Boolean;
begin
  // 수정이나 삽입 작업이 진행중이 아닌 상태, 즉 레코드를 조회하는
```

```
  // 상태에서만 자식 폼의 변경을 가능하게 한다.
  Result := FormMode = fmBrowse;
end;

function TPartsForm.GetFormMenu: TMainMenu;
begin
 // 자식 폼의 메인 메뉴를 돌려준다. 메인 폼은 이 메소드가 돌려준 메뉴를
 // 자신의 메인 메뉴에 합친다.
  Result := mmFormMenu;
end;

procedure TPartsForm.sbFindClick(Sender: TObject);
begin
  // 부품 번호로 부품을 검색한다.
  inherited;
  DDGSalesDataModule.SearchForPart;
end;

procedure TPartsForm.sbBrowseClick(Sender: TObject);
begin
  // 현재 레코드에 가해진 변경들을 취소한 후
  // 폼을 조회 상태로 설정한다.
  inherited;
  if not (FormMode = fmBrowse) then
    DDGSalesDataModule.CancelPart;
end;

 end.
```

목록을 보면 TCustomerForm과 거의 동일하다는 점을 알 수 있을 것이다. 4장에서 만든 프레임웍과 이번 장에서 만든 데이터 모듈의 설계가 제대로 되어 있기 때문에 이러한 일관성이 생겼다고 할 수 있다.

TSalesForm - 판매 조회

그림 31.6은 기존의 판매를 조회할 때 사용하는 판매 폼의 모습이다. 이 폼의 소스 코드에는 GetFormMenu()라는 메소드 밖에 없다. 이 폼은 메인 폼의 메뉴에 합칠 메뉴가 없기 때문에 GetFormMenu() 메소드는 무조건 nil을 돌려주도록 재정의되었다. 그 외에는 별 달리 설명할 것이 없으므로 소스 코드는 제시하지 않겠다. 이 폼은 부록 CD의 SalesFrm.pas에 해당한다.

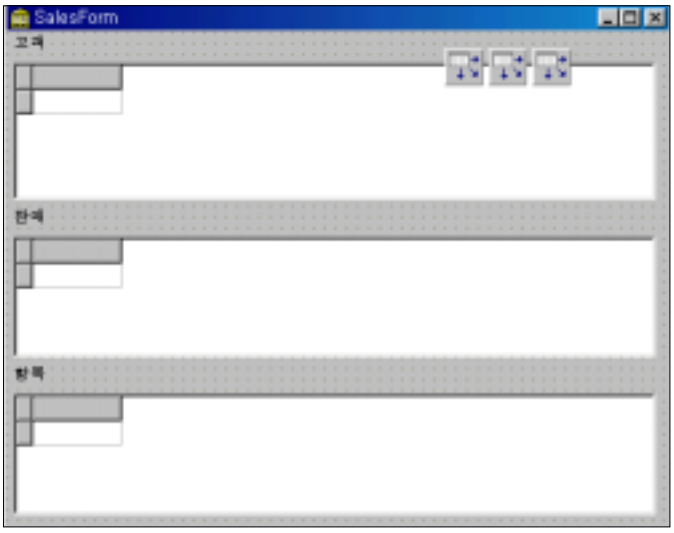


그림 31.6
TSalesForm - 판매 조회 폼

TNewSalesForm - 판매 입력 폼

TNewSalesForm은 네 개의 자식 폼들 중 가장 복잡하다. 이 폼의 소스 코드는 목록 31.8에 나와 있다. 가장 복잡한 폼이긴 하지만, 별로 어렵지는 않다. 코드의 주석을 보면 쉽게 이해할 수 있을 것이다. TDBNavStatForm을 상속하는 다른 자식 폼들과는 달리, 이 폼은 TChildForm을 직접 상속하기 때문에 TToolBar 컴포넌트를 부모 컴포넌트에게 돌려주는 메소드가 정의되어 있지 않다. 그래서 SetToolBarParent()라는 메소드를 직접 정의했다. 그림 31.7은 TNewSalesForm의 모습이다.

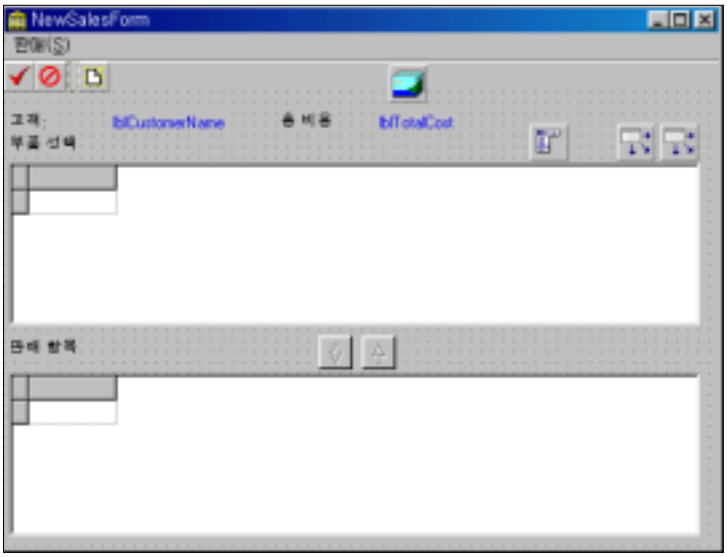


그림 31.7
TNew SalesForm - 새 판매 데이터 입력 폼

목록 31.8 Ch31\NewSalesFrm.pas

```
unit NewSalesFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, CHILDFRM, Grids, DBGrids, Buttons, StdCtrls, Db, Menus,
  ToolWin, ComCtrls, ImgList;

type
  TNewSalesForm = class(TChildForm)
    dsParts: TDataSource;
    dsTempItems: TDataSource;
    lblCustomer1: TLabel;
    lblCustomerName: TLabel;
    lblTotCost: TLabel;
```

```
    lblTotalCost: TLabel;
    lblSelectParts: TLabel;
    sbAddPart: TSpeedButton;
    sbRemovePart: TSpeedButton;
    lblSaleItems: TLabel;
    dbgParts: TDBGrid;
    dbgSaleItems: TDBGrid;
    mmFormMenu: TMainMenu;
    mmiSales: TMenuItem;
    mmiNew: TMenuItem;
    mmiCancel: TMenuItem;
    mmiSave: TMenuItem;
    tbSales: TToolBar;
    sbAccept: TToolButton;
    sbCancel: TToolButton;
    tb1: TToolButton;
    sbInsert: TToolButton;
    ilNavigationBar: TImageList;
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure FormShow(Sender: TObject);
    procedure sbAddPartClick(Sender: TObject);
    procedure mmiNewClick(Sender: TObject);
    procedure mmiCancelClick(Sender: TObject);
    procedure mmiSaveClick(Sender: TObject);
  private
    AddingSale: Boolean;
    function CanChange: Boolean; override;
    procedure SetSaleMenus;
    procedure TempItemsAfterChange(DataSet: TDataSet);
  public
    function GetFormMenu: TMainMenu; override;
    procedure SetToolBarParent(AParent: TWinControl);
  end;

var
  NewSalesForm: TNewSalesForm;

implementation
```

```
uses SalesDM;

{$R *.DFM}

function TNewSalesForm.CanChange: Boolean;
begin
  Result := not AddingSale;
end;

procedure TNewSalesForm.FormCreate(Sender: TObject);
begin
  inherited;
 // DDGSalesDataModule의 tblTempItems 테이블을 연다.
  DDGSalesDataModule.OpenTempItems;
  AddingSale := False; // 초기에는 판매를 추가하는 상태가 아니다.

 { TempItemsAfterChange() 메소드를 DDGSalesDataModule의
     AfterTempItemsChange 이벤트 처리 메소드로 설정한다. }
  DDGSalesDataModule.AfterTempItemsChange := TempItemsAfterChange;
  SetSaleMenus;
end;

procedure TNewSalesForm.FormDestroy(Sender: TObject);
begin
  // DDGSalesDataModule.tblTempItems 테이블을 닫는다.
  inherited;
  DDGSalesDataModule.CloseTempItems;
end;

procedure TNewSalesForm.FormShow(Sender: TObject);
begin
 // 현재 고객의 이름을 얻는다.
  inherited;
  lblCustomerName.Caption := DDGSalesDataModule.GetCustomerName;
 { 폼이 방금 떴으므로 판매액은 0이어야 한다. }
  lblTotalCost.Caption    := '$   0.00';
end;
```

```
procedure TNewSalesForm.TempItemsAfterChange(DataSet: TDataSet);
begin
 // 사용자가 어떤 변경을 할 때마다 폼에 나타나는 판매 총액을 갱신해야 하므로
 // 데이터 모듈의 tblTempItems가 발생시키는 AfterPost 이벤트를 적절히
 // 처리해야 한다. 이 메소드가 그런 작업을 한다.
  lblTotalCost.Caption := FormatFloat('$#,##0.00',
      DDGSalesDataModule.SaleItemsTotalPrice);
end;

procedure TNewSalesForm.sbAddPartClick(Sender: TObject);
begin
  // 선택된 항목을 판매에 추가한다.
  inherited;
  DDGSalesDataModule.AddItemToSale;
end;

procedure TNewSalesForm.mmiNewClick(Sender: TObject);
begin
  // 폼을 판매 추가 작업 상태로 만든다.
  inherited;
  AddingSale := True;
  SetSaleMenus;
end;

procedure TNewSalesForm.mmiCancelClick(Sender: TObject);
begin
  // 현재 판매를 취소시킨다.
  inherited;
  AddingSale := False;
  DDGSalesDataModule.CancelSale;
  SetSaleMenus;
end;

procedure TNewSalesForm.mmiSaveClick(Sender: TObject);
begin
  // 현재 판매를 저장한다.
  inherited;
  DDGSalesDataModule.SaveSale;
  AddingSale := False;
```

```
  SetSaleMenus;
 { TempItemsAfterChange() 이벤트 처리 메소드를 호출해서 폼의 컨트롤들을
   적절히 갱신한다. }
  TempItemsAfterChange(nil);
end;

procedure TNewSalesForm.SetSaleMenus;
begin
  // 폼의 상태에 맞게 메뉴 항목들과 버튼들을 적절히 활성화/비활성화시킨다.
  mmiNew.Enabled       := not AddingSale;
  mmiCancel.Enabled    := AddingSale;
  mmiSave.Enabled      := AddingSale;
  sbAddPart.Enabled    := AddingSale;
  sbRemovePart.Enabled := AddingSale;

  sbAccept.Enabled     := mmiSave.Enabled;
  sbCancel.Enabled     := mmiCancel.Enabled;
  sbInsert.Enabled     := mmiNew.Enabled;

end;

function TNewSalesForm.GetFormMenu: TMainMenu;
begin
 // 자식 폼의 메인 메뉴를 돌려준다. 메인 폼은 이 메소드가 돌려준 메뉴를
 // 자신의 메인 메뉴에 합친다.
  Result := mmFormMenu;
end;

procedure TNewSalesForm.SetToolBarParent(AParent: TWinControl);
begin
 { 이 폼의 툴바를 부모에게 돌려준다. 이 폼이 TDBNavStatForm을 상속했다면
   이 메소드가 이미 존재했겠지만, 이 폼은 TChildForm을 상속한 것이기 때문에
   이 메소드를 직접 작성해야 한다. }
  tbSales.Parent := AParent;
end;

 end.
```

고객 검색 대화상자

TCustomerSearchForm은 DDGSalesDataModule이 CUSTOMER 테이블에 대한 검색을 수행할 때 사용할 SQL 질의문을 제공하는 역할을 한다. 이 폼은 사용자가 입력한 값들을 이용해서 SQL 질의문을 생성해 낸다. 그림 31.8은 고객 검색 폼인 TCustomerSearchForm이다.

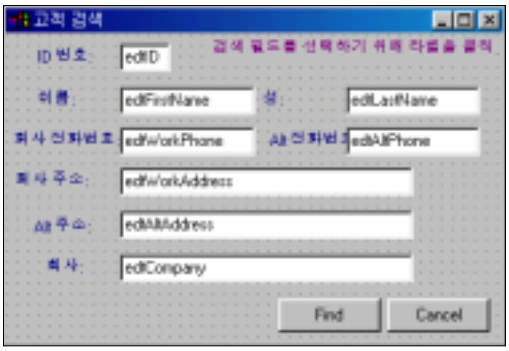


그림 31.8
고객 검색 폼

TCustomerSearchForm은 자식 폼으로 사용되는 것이 아니라 그냥 대화상자로 사용되는 것이며, 데이터 인식 컨트롤들은 가지고 있지 않다. 사용자는 이 폼의 여러 에디트 필드들에 검색 조건으로 사용할 값들을 입력한다. 그리고 반드시 필드 이름에 해당하는 레이블(TLabel)들을 클릭해서 검색할 필드를 선택해주어야 한다(여러 개 선택할 수 있다). 레이블을 클릭하면 그 레이블의 색은 crRed로 바뀐다. Find 버튼을 클릭하면 입력된 값들과 선택된 필드들에 기초해서 SQL 질의문이 만들어진다.

목록 31.9는 TCustomerSearchForm의 소스 코드이다.

목록 31.9 Ch31\CustomerSearchFrm.pas

```
unit CustomerSrchFrm;

interface

uses WinTypes, WinProcs, Classes, Graphics, Forms, Controls, Buttons,
  StdCtrls, SysUtils;
```

```
type
  TCustomerSearchForm = class(TForm)
    lblIDNumber: TLabel;
    edtIDNumber: TEdit;
    lblFirstName: TLabel;
    lblLastName: TLabel;
    lblAltPhone: TLabel;
    lblWorkPhone: TLabel;
    lblWorkAddress: TLabel;
    lblAltAddress: TLabel;
    lblCompany: TLabel;
    edtFirstName: TEdit;
    edtLastName: TEdit;
    edtWorkPhone: TEdit;
    edtAltPhone: TEdit;
    edtWorkAddress: TEdit;
    edtAltAddress: TEdit;
    edtCompany: TEdit;
    btnCancel: TButton;
    btnFind: TButton;
    lblInstruction: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure FindCustBtnClick(Sender: TObject);
    procedure CancelBtnClick(Sender: TObject);
    procedure lblIDNumberClick(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
  private
    FindPressed: Boolean;
    procedure ClearEditFields;
    function BuildSQLStatement: string;
  public
    QueryString: String;
  end;

function SearchCustomer: String;

implementation
```

```
{$R *.DFM}

uses Dialogs;

function SearchCustomer: String;
var
  CustomerSearchForm: TCustomerSearchForm;
begin
  Result := EmptyStr;
  CustomerSearchForm := TCustomerSearchForm.Create(Application);
  try
    if CustomerSearchForm.ShowModal = mrOk then
      Result := CustomerSearchForm.QueryString;
  finally
    CustomerSearchForm.Free;
  end;

end;


function TCustomerSearchForm.BuildSQLStatement: string;
{ 이 메소드는 사용자가 지정한 검색 필드들에 기반해서 SQL 질의문을 생성해 낸다.
  사용자가 지정한 검색 필드들에 해당하는 레이블은 색이 clRed이므로 각 레이블의
  색을 검사하면 사용자가 어떤 필드들을 선택했는지 알 수 있다. 선택된 필드를
  판별한 다음, 그 필드에 입력된 값을 이용해서 적절한 SQL 질의문을 만들어낸다. }
var
  Sep: String[3]; // SQL 질의문의 분리자로 사용할 문자열
begin
  Sep := '';
  Result := '';

  if lblIDNumber.Font.Color = clRed then
  begin
    Result := Format('(CUSTOMER_ID = %s)', [edtIDNumber.Text]);
    Sep := 'AND';
  end;
```

```
if lblLastName.Font.Color = clRed then
begin
  Result := Format('%s %s (UPPER(LNAME) = "%s")',
                [Result, Sep, UpperCase(edtLastName.Text)]);
  Sep := 'AND';
end;

if lblFirstName.Font.Color = clRed then
begin
  Result := Format('%s %s (UPPER(FNAME) = "%s")',
                [Result, Sep, UpperCase(edtFirstName.Text)]);
  Sep := 'AND';
end;

if lblWorkPhone.Font.Color = clRed then
begin
  Result := Format('%s %s (UPPER(WORK_PHONE) = "%s")',
                [Result, Sep, UpperCase(edtWorkPhone.Text)]);
  Sep := 'AND';
end;

if lblAltPhone.Font.Color = clRed then
begin
  Result := Format('%s %s (UPPER(ALT_PHONE) = "%s")',
                [Result, Sep, UpperCase(edtAltPhone.Text)]);
  Sep := 'AND';
end;

if lblWorkAddress.Font.Color = clRed then
begin
  Result := Format('%s %s (UPPER(WORK_ADDRESS) = "%s")',
                [Result, Sep, UpperCase(edtWorkAddress.Text)]);
  Sep := 'AND';
end;

if lblAltAddress.Font.Color = clRed then
begin
  Result := Format('%s %s (UPPER(ALT_ADDRESS) = "%s")',
```

```
                  [Result, Sep, UpperCase(edtAltAddress.Text)]);
    Sep := 'AND';
  end;

  if lblCompany.Font.Color = clRed then
  begin
    Result := Format('%s %s (UPPER(COMPANY) = "%s")',
                  [Result, Sep, UpperCase(edtCompany.Text)]);
  end;

  if Length(Result) > 0 then
    Result := Format('SELECT CUSTOMER_ID FROM CUSTOMER WHERE (%s)',
      [Result]);

end;

procedure TCustomerSearchForm.ClearEditFields;
{ 이 메소드는 모든 에디트 필드들을 비우고, 해당 레이블들의 색을 clNavy로
  설정한다. }
var
  i: word;
begin
  for i := 0 to ComponentCount - 1 do
  begin
    if Components[i] is TEdit then
      TEdit(Components[i]).Text := '';

    if Components[i] is TLabel then
      TLabel(Components[i]).Font.Color := clNavy;
  end;

end;

procedure TCustomerSearchForm.FormCreate(Sender: TObject);
begin
  FindPressed := False;
  // 에디트 필드들을 비운다.
  ClearEditFields;
```

```
end;

procedure TCustomerSearchForm.FindCustBtnClick(Sender: TObject);
begin
  FindPressed := True;
  // SQL 질의문을 QueryString에 넣는다. 이 대화상자를 호출한 쪽에서는
  // QueryString을 통해서 SQL 질의문을 얻게 된다.
  QueryString := BuildSQLStatement;
end;

procedure TCustomerSearchForm.CancelBtnClick(Sender: TObject);
begin
  ClearEditFields;
end;

procedure TCustomerSearchForm.lblIDNumberClick(Sender: TObject);
{ 모든 레이블들은 이 OnClick 이벤트 처리 메소드에 연결되어 있다. 이 메소드는
  레이블의 색을 적절히 바꾸는 역할을 한다. clRed는 해당 필드가 검색 조건에
  포함된다는 뜻이다. }
begin
  with (Sender as TLabel) do
    if Font.Color = clNavy then
      Font.Color := clRed
    else
      Font.Color := clNavy;
end;

procedure TCustomerSearchForm.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
 { 폼을 닫기 전에 사용자가 검색 필드들을 선택했는지 확인한다. }
  if (QueryString = '') and FindPressed then
  begin
    MessageDlg('라벨을 클릭함으로써 검색 필드를 선택해야만 합니다.',
               mtInformation, [mbOk], 0);
    Action := caNone;
  end
  else begin
```

```
      Action := caHide;
      ClearEditFields;
    end;
  end;

  end.
```

이 폼의 핵심적인 메소드는 SQL 질의문을 만들어내는 BuildSQLStatement() 함수이다. 이 메소드는 각 레이블들의 색을 검사해서 레이블의 색이 clRed이면 그 레이블에 해당하는 에디트 필드의 값을 Format() 함수에 적용해서 SQL 질의문을 만들어낸다.

ClearEditFields()는 모든 레이블들의 색을 clNavy로 설정하고 해당 에디트 컨트롤들을 비운다. 이 메소드는 FormCreate() 이벤트 처리 메소드에서 폼이 생성될 때 호출된다.

FormClose() 이벤트 처리 메소드는 QueryString이 빈 문자열인지 검사한다. QueryString이 빈 문자열이면 사용자가 아직 검색 필드들을 선택하지 않은 것이므로 사용자에게 적절한 메시지 상자를 표시한다. 만일 사용자가 Cancel 버튼을 클릭해서 폼이 닫히는 것이라면 QueryString에 관계없이 무조건 폼을 닫는다. Find 버튼이 클릭되면 Boolean 변수 FindPressed가 True가 되므로 FindPressed가 False이면 Cancel 버튼이 클릭된 것이다.

Find 버튼이 클릭되면 호출한 폼은 SQL 질의문을 얻게 된다.

요약

이렇게 해서 재고 관리 애플리케이션이 완성되었다. 이번 장에서는 2-티어 구조(two-tier) 클라이언트/서버 애플리케이션을 설계하는 방법을 제시했다. 2-티어 구조 모델은 가장 대표적인 클라이언트/서버 시스템의 구조라고 할 수 있다. 그러나 인터넷이나 관련 기술 분야에서는 3-티어 구조 모델이 점점 더 인기를 끌고 있는데, 6부에서는 3-티어 모델에 기반한 MIDAS 애플리케이션을 만들어 보겠다.

쉬어가는 페이지