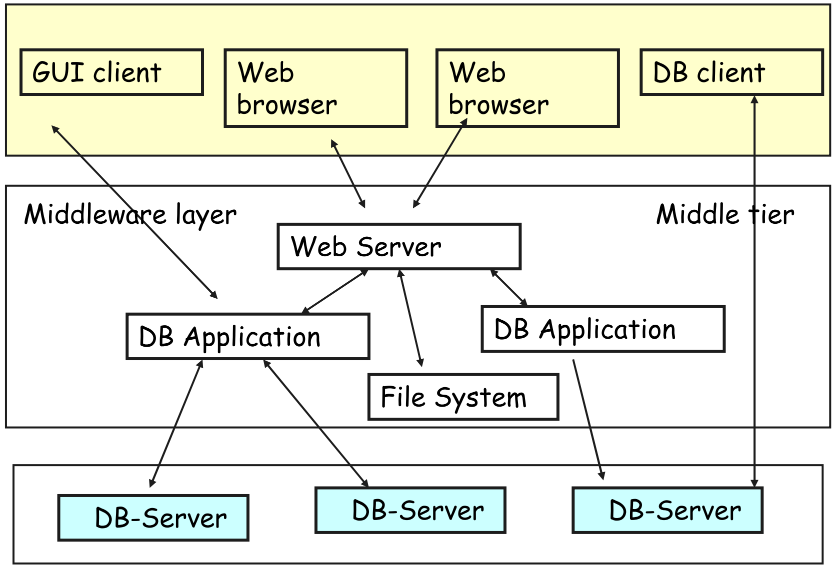
12 Embedding SQL in Programming languages

Kemper / Eickler: chap. 4.19-4.23;
Melton: chap. 12,13,17-19, Widom, Ullman, Garcia-Molina: chapt.8
Christian Ullenboom Java ist auch eine Insel, Kap. 20, Galileo Comp.

Using SQL from Programs Introduction

- SQL is a data sublanguage
- Needs a host language
  - Control structures
  - User interface: output formatting, forms
  - Transactions: more than one DB interaction as a unit of work
- Issues
  - Language mismatch ("impedance mismatch")
    - Set oriented operations versus manipulation of individuals
    - How to interconnect program variables and e.g attributes in SQL statements?
    - Should an SQL-statement as part of a program be compiled, when?
- Question: could you imagine a language bringing both worlds together?

Three-tier architecture (example)



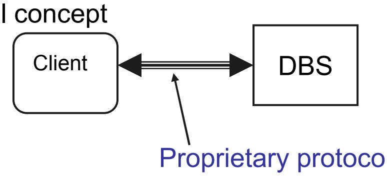
Using SQL from Programs Introduction

Overview of language / DB integration concepts

- "Fourth Generation Languages"
- Module Language --> PSM (~ PL/SQL, PLpgSQL)
  - Standardized in SQL-99
- Interface of standard programming languages
  - Call level interface, proprietary library routines, API
    Standardized: SQL CLI Open Database connection (ODBC),
  - Embedded C / Java / ..
    Standardized language extensions
  - Standardized API
    Java DBC "Fourth generation Language"
- Stored Procedures
  - C / Java / Perl / Python, ...
- Component architectures: hiding the details of DB interaction, Enterprise Java Beans (EJB)

SQL from Programs "4. Generation Languages"

- Proprietary "Fourth generation language (4GL)"
  - Underlying assumption:
    - most application programs are algorithmically simple
    - sophisticated output formatting needed
    - it should be difficult for users to switch from one DBS to another
  - Technical concept



  - Client evolved from simple Terminal to 4GL-Interpreter
  - Open systems movement and HTTP / HTML / Java makes 4GL less important

Using SQL from Programs Modules

- Standardization efforts (SQL 89 / SQL–99) Modules and Embedded SQL
  - SQL Modules
    - Separate parameterized Modules of SQL statements
    - Compiled for a particular language (e.g. COBOL, C, ADA...)
    - Linked to application program (statically?)
    - Disadvantage
      - SQL code hidden in application and vice versa
      - Not widely used
    - Superseded by flexible stored procedure concept

Using SQL from Programs Call interface

- Call level interface
  - Language and DBS specific library of procedures to access the DB
  - Example: MySQL C API
    - Communication buffer for transfering commands and results
    - API data types like
      `MYSQL` handle for db connections
      `MYSQL_RES` structure which represents result set
    - API functions
      `mysql_real_query()`
      `mysql_real_query (MYSQL *mysql, const char * query, unsigned int length)`
      query of `length` of character string in buffer
      and many more....
  - Standard : Open Database Connection (ODBC)
  - Predecessor of Java Database Connection (JDBC), see below

SQL Call level interface (SQL/CLI)

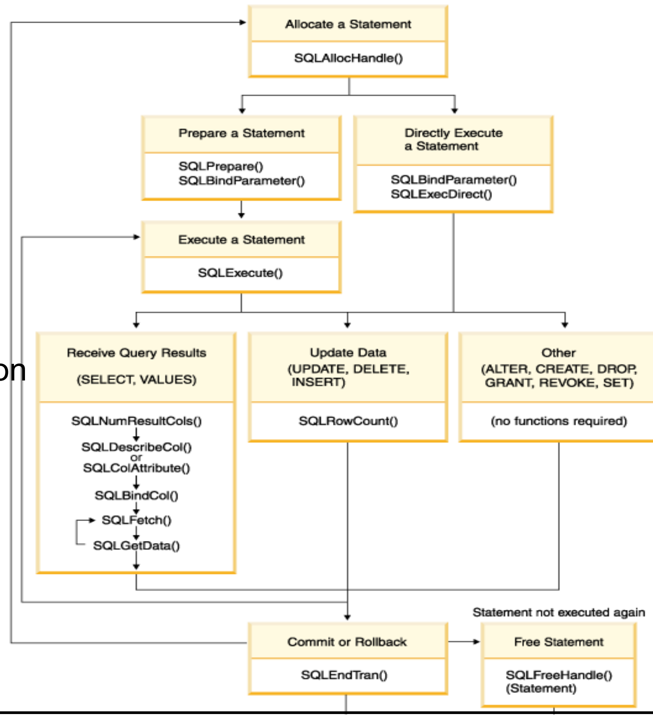
- Standardized Interface to C / C++ defined by X/OPEN and SQL Accesss group
- Main advantages
  - DBS-independent
  - Application development independent from DBS (as opposed to Embedded SQL precompiler approach, see below)
  - Easy to connect to multiple DB
- Microsoft implementation
  ODBC (= Open Database Connectivity) de facto standard, available not only for MS products

Main cycle of transaction execution with SQL/CLI

Calls are embedded in the application program

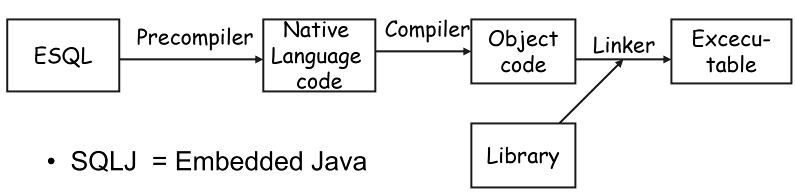
See also JDBC , ESQL

source:
IBM DB2 manual



12.2 Embedded SQL

- Embedded SQL – the most important(?) approach
  - Concepts
    - Program consists of "native" and SQL-like statements
    - Precompiler compiles it to native code, includes calls to DBS resources
    - Employs call level interface in most implementations
    - Most popular: Embedded C   (Oracle: PRO*C)



- SQLJ = Embedded Java

Embedded SQL (ESQL) Syntax and more

- Well defined type mapping (for different languages)
- Exception handling (`WHENEVER condition action`)
  `SQLSTATE, SQLCODE` (deprecated)
- Syntax for embedded SQL statements
- Binding to host language variables

hypothetical syntax, like SQLJ

```
#sql  {SELECT m# FROM M
         WHERE titel = :titleString};}...
#sql {FETCH ...INTO :var1}
```


ESQL

- C / Java embedding

  – ESQL/C

```
EXEC SQL UPDATE staff SET job = 'Clerk'
   WHERE job = 'Mgr';
 if ( SQLCODE < 0  printf( "Update Error: ... );
```

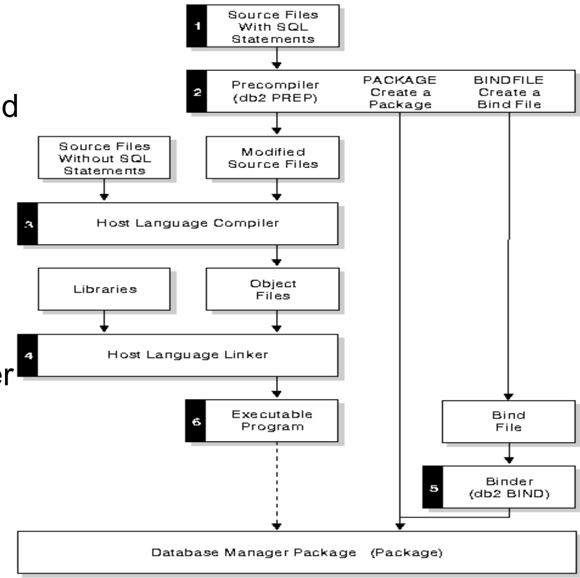
  – SQLJ

```
try { #sql { UPDATE staff SET job = 'Clerk'
      WHERE job = 'Mgr' }; }
catch (SQLException e)
 { println( "Update Error: SQLCODE = " + ... );
```

ESQL code generation

Code generated basically at compile time.

DBS and DB must be known before runtime in order to generate executables



from:
DB2 manual

12.2.1 ESQL Static / dynamic embedding

Static versus dynamic SQL:

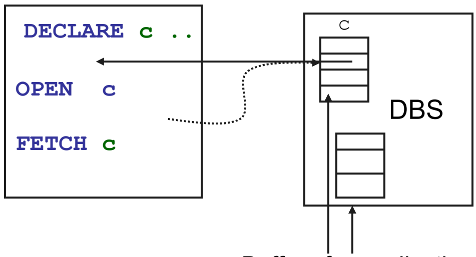
- Static: all SQL commands are known in advance, SQL-compilation and language binding at precompile time

- Dynamic
  - (i) SQL-String executed by DBS:
    Operator tree, optimization, code binding....
  - (ii) SQL-String *prepared* (compiled) at runtime.
    Performance gain in loops etc.

12.2.2 ESQL Cursors

Cursor concept

- How to process a result set one tuple after the other?
- CURSOR: name of an SQL statement and a handle for processing the result set record by record
- Cursor is defined, opened at runtime (= SQL-statement is excecuted) and used for FETCHing single result records



Cursor concept used in most language embeddings of SQL (ESQL-C, PL/SQL, JDBC and more)

Buffers for application program cursors
DBS may determine result set in a lazy or eager way

ESQL Cursors

- Explicit cursors: Declared and named by the programmer
  - Sometimes implicit cursors for individual SQL statements are used in 4GL
- Cursor
  - assigns a name to an SQL statement.
    Cursor / SQL statement do not bind the result attributes to variables
  - allows to traverse the result set (the "active set") row by row

```
Declare curs for Select c#, lname, m.title
             from C, R, M where  ....
```

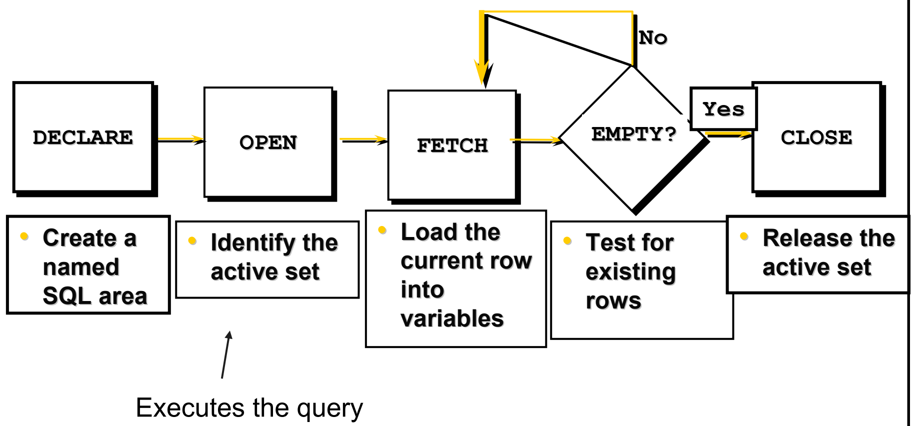
Active set

7369	SMITH	To be or ..
7566	JONES	Metropolis
7788	SCOTT	Forest Gump
7876	ADAMS	Forest Gump
7902	FORD	Star Wars I

Cursor curs → Current row

ESQL Cursors

- Controlling a cursor: the necessary steps



ESQL Cursors

- Opening

```
OPEN  cursor_name;
```

In a compiled language environment (e.g. embedded C):

- bind input variables
- execute query
- put (first) results into communication (context) area
- no exception if result is empty
  has to be checked when fetching the results
- positions the cursor before the first row of the result set (" –1 ")

First steps in an interpreted language (e.g. 4GL PL/SQL) :

- allocate context area
- parse query

ESQL Cursors

- Fetch

```
FETCH curs INTO :x, :nameVar, :titleVar;
```

Cursor scrolling (Declare c SCROLL cursor.. in SQL 92):

FETCH [NEXT | PRIOR | FIRST | LAST |
[ABSOLUTE | RELATIVE expression]]
FROM cursor INTO target-variables

```
FETCH curs PRIOR INTO :x, :nameVar, :titleVar;
```

=

```
FETCH curs RELATIVE -1  INTO :x, :nameVar, :titleVar;
```

Single row SELECT does not need a FETCH but result is bound to variables: SELECT a,b FROM... INTO :x,:y WHERE

12.2.3 ESQL

```
#include <stdio.h>

/* declare host variables
*/
char userid[12] =
"ABEL/xyz";
char emp_name[10];
int  emp_number;
int  dept_number;
char temp[32];
void sql_error();

/* include the SQL
Communications Are
*/ #include <sqlca.h>
```

```
main()
{  emp_number = 7499;
/* handle errors */
   EXEC SQL WHENEVER SQLERROR
    do sql_error("Oracle error");

/* connect to Oracle */
   EXEC SQL CONNECT :userid;
   printf("Connected.\n");

/* declare a cursor */
   EXEC SQL DECLARE emp_cursor
            CURSOR FOR
    SELECT ename
       FROM emp
       WHERE deptno =
              :dept_number;
```

Establish DB connection

ESQL Example: Embedded C

```
printf("Department number? ");
   gets(temp);
   dept_number = atoi(temp);
/* open the cursor and identify the active
set */
   EXEC SQL OPEN emp_cursor; …
/* fetch and process data in a loop
   exit when no more data */
   EXEC SQL WHENEVER NOT FOUND DO break;
   while (1)
   {EXEC SQL FETCH emp_cursor INTO
   :emp_name; ..
}
   EXEC SQL CLOSE emp_cursor;
   EXEC SQL COMMIT WORK RELEASE;
   exit(0); }
```

Close cursor before another SQL statement is executed

ESQL Exception handling

- Exception handling

```
void  sql_error(msg)
char *msg;
{
    char buf[500];
    int buflen, msglen;
    EXEC SQL WHENEVER
             SQLERROR CONTINUE;
    EXEC SQL ROLLBACK WORK
                     RELEASE;
    buflen = sizeof (buf);
    sqlglm(buf, &buflen, &msglen);
    printf("%s\n", msg);
    printf("%*.s\n", msglen, buf);
    exit(1);
```

ESQL Exception handling

```
EXEC SQL WHENEVER SQLERROR GOTO sql_error;
...
sql_error:
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK WORK RELEASE;
  ...
```

Without the WHENEVER SQLERROR CONTINUE statement, a ROLLBACK error would invoke the routine again, starting an infinite loop.

12.2.4 Positioned Update

- Update / Delete statements in general use search predicates to determine the rows to be updated

```
Update M
  set price_Day = price_Day+1 where price_Day <= 1
```

- Often useful: step through a set of rows and update some of them ⇨ positioned update

```
DECLARE myCurs  FOR SELECT ppd, title FROM M
     FOR UPDATE ON ppd
UPDATE M SET ppd = ppd + 1
    WHERE CURRENT OF  myCurs    /* delete in a
                                /*similar way
```

Caveat: Use the capabilities of SQL!

It would be stupid to check a predicate on a row within the FETCH loop and then update the row.

- A cursor may declared FOR READ ONLY (which basically results in some performance gains)

ESQL Cursor sensitivity

Which state has the database during processing?

```
EXEC SQL DECLARE myCurs  FOR SELECT price_Day, title
  FROM M  FOR UPDATE ON price_Day
               WHERE price_Day < 2
EXEC SQL OPEN...
...
EXEC SQL FETCH myCurs INTO .....
UPDATE M SET price_Day = price_Day + 2
          WHERE CURRENT OF  myCurs /* similar for
                                   /* delete
```

Is the row under the cursor still in the result set?

Yes and No !

- A cursor declared INSENSITIVE does not make visible any changes (update, delete) until the cursor is closed and reopened.

12.3 Transactions in application programs

12.3.1 Definition

– Sequence of operations on DB which form a "unit of work"

– Example: Bank account transfer ("debit / credit') :

```
read (acc1); read (acc2);
acc1=acc1-amount ; acc2 = acc2+ amount;
write(acc1); write (acc2);
```

– System must guarantee "correct execution"

– "Dependable system"

dependable: verlässlich, betriebssicher, zuverlässig

Transaction braces

.

TA Syntax :
Every operation on DB between the beginning of the sequence of operations and a
COMMIT WORK or
ROLLBACK WORK

No explicit "transaction begin" command needed

```
... OPEN MyCurs;........ ; COMMIT;  OPEN ...
```

Beginning of first TA
(first SQL command in program)

end of first TA, beginning of next TA

But SQL-3: **START TRANSACTION**, **Postgres: BEGIN**

Transaction semantics

Transactional semantics means:

DBS guarantees certain executional properties

- "All or nothing" semantics — ATOMICITY
  - All effects are made permanent at COMMIT, not before .
    TA has no effect after ROLLBACK
- "Now and forever" — DURABILITY
  - DBS guarantees the effects after COMMIT has been processed successfully
- "Solve concurrency conflicts" — ISOLATION
  - Conflict resolution of concurrent operations on DB
- "Keep consistent DB consistent" — CONSISTENCY
  - Preservation of integrity

Transactions

- How does DB System guarantee the properties?
  - ⇨ Implementation of DBS
- Application programming with transaction
  - Syntactically mark unit of work:

    START TRANSACTION ……. COMMIT;

    or:

    START TRANSACTION …….
    IF (everyThingOK) COMMIT
    ELSE ROLLBACK; ENDIF – no effect
  - exception handling if application commits but DBS cannot guarantee
  - Isolation levels

12.3.2 Isolation

- Important task of transaction manager: isolate concurrent users from each other

```
SELECT balance INTO :myVar
FROM account
WHERE acc# = :myAcc;
If myVar + dispo – amount >=0
 UPDATE account SET
   balance = myVar – amount
 WHERE acc# = :myAcc;
Call ATM_pay_out;
ENDIF;
COMMIT;
```

```
…
SELECT SUM(balance),owner

FROM account
GROUP BY owner;
COMMIT;
DBS_OUTPUT.PutLine(…);
```

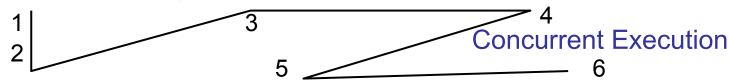
concurrent execution in independent DB sessions

Conflict? Not a big deal in this case,
but may be SUM is incorrect.

Isolation

Worst case: lost update

T1: progVar← read(x); progVar++; write (x ← progVar)



T2: progVar ← read(x); progVar++; write (x ← progVar)

Read of T1 and T2: x=7; Increment by T1: x== 8, increment by t2: x==8

Lost update: two independent updaters update the same object. Conflict may result in a wrong value! Updates is lost!
Not allowed in any serious multiuser DBS

Isolation levels : control behaviour of transaction

- No problem at all if only READs
- How much isolation does a TA need?
  - Application dependent: is it acceptable that the balance per customer does not reflect the correct balances of her account?
- read / write ratio?
- What is the conflict probability ?

Isolation level:

The kind of conflicts a program is willing to accept

The more isolation the less parallelism

Transactions in application programs

- Isolation Levels

  Suppose TA1 decreases the prices of some movies in the movie DB by 5%

  TA2 scrolls through all movies

  - Question: does TA2 "see" the new values before TA1 commits?

  READ UNCOMMITTED

  - Yes: updates of TA1 are immediately visible
    but only if TA2 has isolation level read uncommitted

  ```
  SET TRANSACTION  READ ONLY,
     ISOLATION LEVEL READ UNCOMMITTED
  ```

  – Lowest locking overhead, but unpleasant effects may happen (Examples?)

  ≡ READ COMMITTED in Postgres

Setting isolation levels

SET TRANSACTION <mode> [,<mode>]$_0^{\,n}$

<mode> = <access mode> |
[ISOLATION LEVEL] <isolation> |
DIAGNOSTIC SIZE <simple_value>

<access mode > = READ ONLY | READ WRITE

<isolation> = READ UNCOMMITTED |
| READ COMMITTED
| REPEATABLE READ
| SERIALIZABLE

Diagnostic: area for details about exceptions, only for ESQL

Different default modes: READ UNCOMMITTED ⇨ READ ONLY
else READ WRITE

Transactions in application programs

READ COMMITTED ("cursor stability")

- No uncommitted update can be seen by any application
- But TA might see different states of the same object

```
TA2 : R (a), x=x+a;...............          R(b); x:=x+b;...
TA1 :          W(b+10); W(a-10);COMMIT;
```

Value of program variablel x does not reflect DB state
because READ is not REPEATABLE

- Conflicts typically solved by locks ("2-phase locking")
- If "Read committed" but no "repeatable read" required : read-only transaction need only short read locks
  ⇨ higher parallelism

Transactions in application programs

- Isolation levels (4)

REPEATABLE READ

- all read / write conflicts prevented, reads repeatable
  Lock synchronization: all locks held until end of TA

but

```
TA2 : R(a), x=x+a...............  R(b), x:=x+b,...
TA1 :          Insert(z); Commit;
       -- TA2: SUM of attribut of relation S,
       -- TA1: inserts a row into S
```

Unpleasant effect: Phantom records

SERIALIZABLE

- repeatable read + phantoms avoided

Transactions

Isolation levels

- first statement within TA
- Be careful with default modes

```
SET TRANSACTION READ WRITE;
SET TRANSACTION ISOLATION LEVEL READ
                      UNCOMMITTED;
```

TA has default access mode of last SET
i.e. READ ONLY (!)

- Read uncommitted dangerous: may cause inconsistencies
- Read committed is the default in some systems (e.g. Oracle)
- Serializable important for high frequent short transactions with many potential conflicts.
- **AUTOCOMMIT**-mode: implicit COMMIT after each SQL-statement

Transaction Rollback / abort

ROLLBACK

- SQL statement like COMMIT
- "backout" of TA, not any effect on the DB
  "all-or-nothing semantics"
- application programmer decides on rollback

Abort

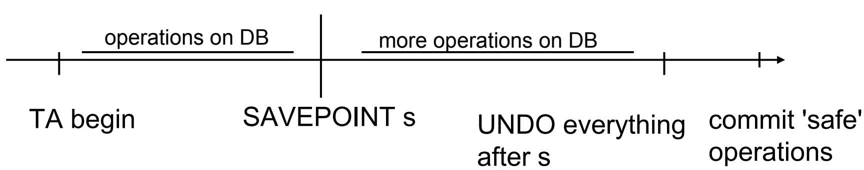
- System kills transaction
- system failure ⇨ user session is aborted ⇨ system recovery
- transaction rollback caused by internal state (e.g. deadlock)
- Recovery of TA by system, of application process control flow by programmer.
  Important: handling of DB exceptions

Deadlock abort detection (Embedd. SQL)

```
#define DEADL_ABORT -60  /* ORA specific
#define TRUE 1
EXEC SQL sql WHENEVER sqlerror CONTINUE;
int count = 0;
while (TRUE) {
    EXEC SQL UPDATE customers
         set discnt = 1.1*discnt WHERE city ='Berlin';
  if (sqlca.sqlcode == DEAD_ABORT) {
     count++;
     if (count < 4) {
      exec sql ROLLBACK;
      } else break;
   else if (sqlca.sqlcode <0) break;
}if (sqlca.sqlcode < 0) {
   print_dberror();
   exec sql rollback;   /* application: go back to start of
   this
   return -1          /*  transaction
} return 0;
```


SAVEPOINTS

- Rollback can be expensive in long TAs
- Use SAVEPOINTs to limit work to be redone



Transaction in applications

- Never have user interaction within a TA
- Resources will be blocked for long time – bad!

```
EXEC SQL SELECT price, quantity into :price, :qoh...
while (TRUE){
  printf("We have %d units... of %d each \n", qoh, price)
  printf ("How many... ",...)    /* check correct input
                                 /* and exit loop
 }
if (qoh >= numberOrdered){
EXEC SQL UPDATE products set quantity = ....
} else ...
EXEC SQL COMMITT;
```

Bad design: resource blocking time depends on user

- How does a better program design look like?