

MACIASZEK, LA. (2001):
Requirements Analysis and System Design.
Developing Information Systems with UML,
Addison Wesley, 378p. (ISBN 0-201-70944-9)

Errata & Addendum

Changes in Chronological Order

DATE OF CHANGE	TYPE OF CHANGE	CHANGE WITH REGARD TO:
30-April-2001	Correction	Chapter 4, p. 142 - Section 4.3.2.3 – Figure 4.13
17-May-2001	Correction	Chapter 8, p. 288 – Section 8.2.2.4 – last sentence on the page
25-May-2001	Addition	Chapter 9 – Section 9.1.3 – new section
30-July-2001	Correction	Chapter 5, p.174
24-October-2001	Extension	Chapter 8, pp.293-294 – Section 8.3.1.4 – entire section (three paragraphs) replaced with an extended material

Chapter

1

Software Process

...

Chapter

2

Underpinnings of Requirements Analysis

...

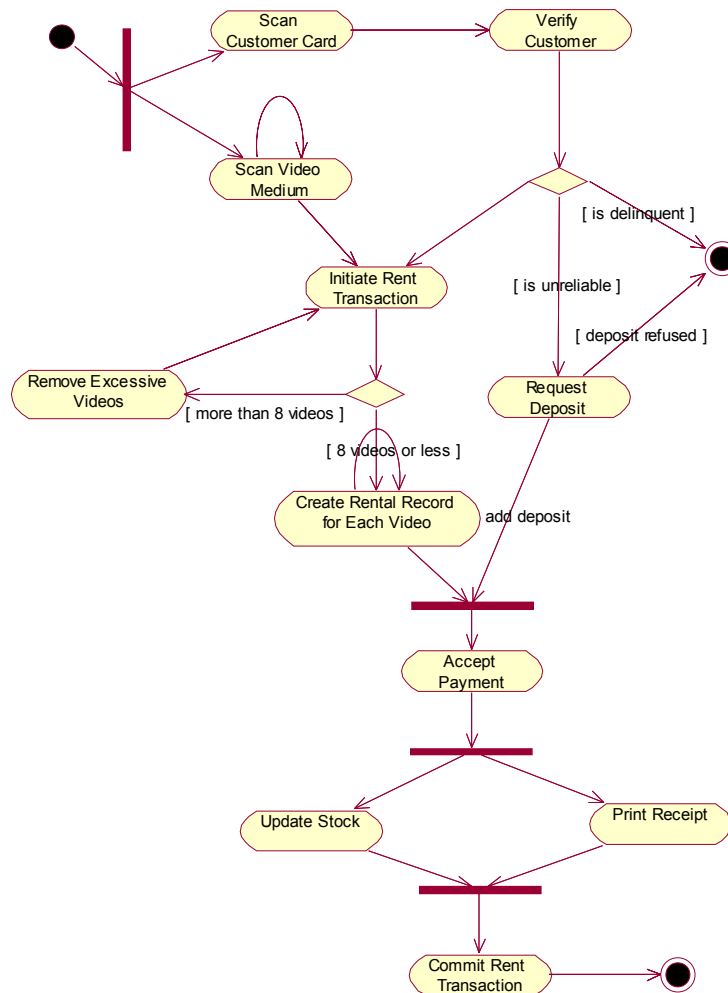
...

p. 142 - Section 4.3.2.3 – Figure 4.13

Replace:

Figure 4.13 (there is a missing connection from the activity “Create rental Record for Each Video” to a synchronization bar that joins to the activity “Accept Payment”)

With:



p. 174 - Section 5.2.3 – last paragraph, last sentence

Replace:

An outer package has access to any classes directly contained in its nested packages.

With:

An inner package has free access to the elements in the outer package but not vice versa. Normal visibility rules apply for accessing elements of an inner package. An inner package is encapsulated.

Chapter

6

Underpinnings of System Design

p. 288 – Section 8.2.2.4 – last sentence on the page**Replace:**

(attributes with private visibility are not inherited)..

With:

(operations with private visibility are not inherited).

p. 293 – Section 8.3.1.4**Replace:**

The whole section (three paragraphs)

With:

A structured type can be used to define a *reference type*. The keyword `ref` is used to define references. For example `emp ref (EmployeeTY)` is a reference in an object table to a structured type. A value of the column `emp` is an OID that identifies an object (a row) in a table of type `EmployeeTY`. We say that it references a row in a *referenceable table*. A referenceable table must be a *typed table*, i.e. a table with an associated *structured type*.

In SQL:1999, the reference types are *scoped* – the table that they reference is known at compilation time. This means that the value `emp` of reference type `ref (EmployeeTY)` must refer to a row in one and only one table (rather than to a row in any table defined on `EmployeeTY`).

For example, we can define a type `TaskTY` with a field `emp` as a reference to type `EmployeeTY`, and a table `Task` of type `TaskTY`, as follows:

```
create type TaskTY
    (description varchar(255) ,
     emp ref(EmployeeTY) scope Employee);

create table Task of TaskTY;
```

An alternative SQL:1999 solution would be to omit the declaration scope `Employee` from the type definition and extend the definition of `create table` as follows:

```
create table Task of TaskTY
    (emp with options scope Employee);
```

In practice, restricting in SQL:1999 a reference type to a single table makes references behave much like foreign keys in RDBs, except that the advantages of using OIDs as reference values still hold. They are unique within the database, never change, are never reused for another object (row), and improve performance when navigating between objects.

Reference types can be used in an ORDB to implement *one-to-one associations*. To implement *many-to-many associations*, **collections** (Section 8.3.1.1) **of references** could be used, if available. Figure 8.10 uses the example presented in Figure 8.3 to show how the many-to-many association between `Student` and `CourseOffering` can be represented in an ORDB. In this and the following examples we use a short form with `table` (instead of `type`) to define references (e.g. `due_emp: ref (Employee)`).

p. 329

Add the following Section before Section 9.2:

9.1.3 Designing Persistency

One of the main issues in client/server program design is the design of persistency for all classes whose instances need to be persistently stored in a database (Section 8.1). On the client end, the design of persistency involves classes in the `Entity Package` and in the `Database Package` (Section 6.1.3.2).

The `Entity Package` contains classes that manage “business objects” in the program’s memory. The names of these classes would normally correspond to the names of persistent “objects” in the database (e.g. relational tables or object tables).

The `Database Package` mediates between in-memory entity objects and persistent “objects” in the database. They establish in effect an Object Storage API that isolates the client program from the persistent storage mechanism. The API can provide a database-independent access to persistent “objects” by encapsulating the database programming language, such as SQL. Such an API can be purchased as a class library from vendors of object databases (ref. Section 8.2) and from other class vendors.

The design of persistency requires a set of classes in the `DatabasePackage` that establish the connection (session) to the database, read (select) records from the database, and modify (insert, delete, update) records in the database. Such classes can be called: `D_Session`, `D_Reader`, and `D_Updater`.

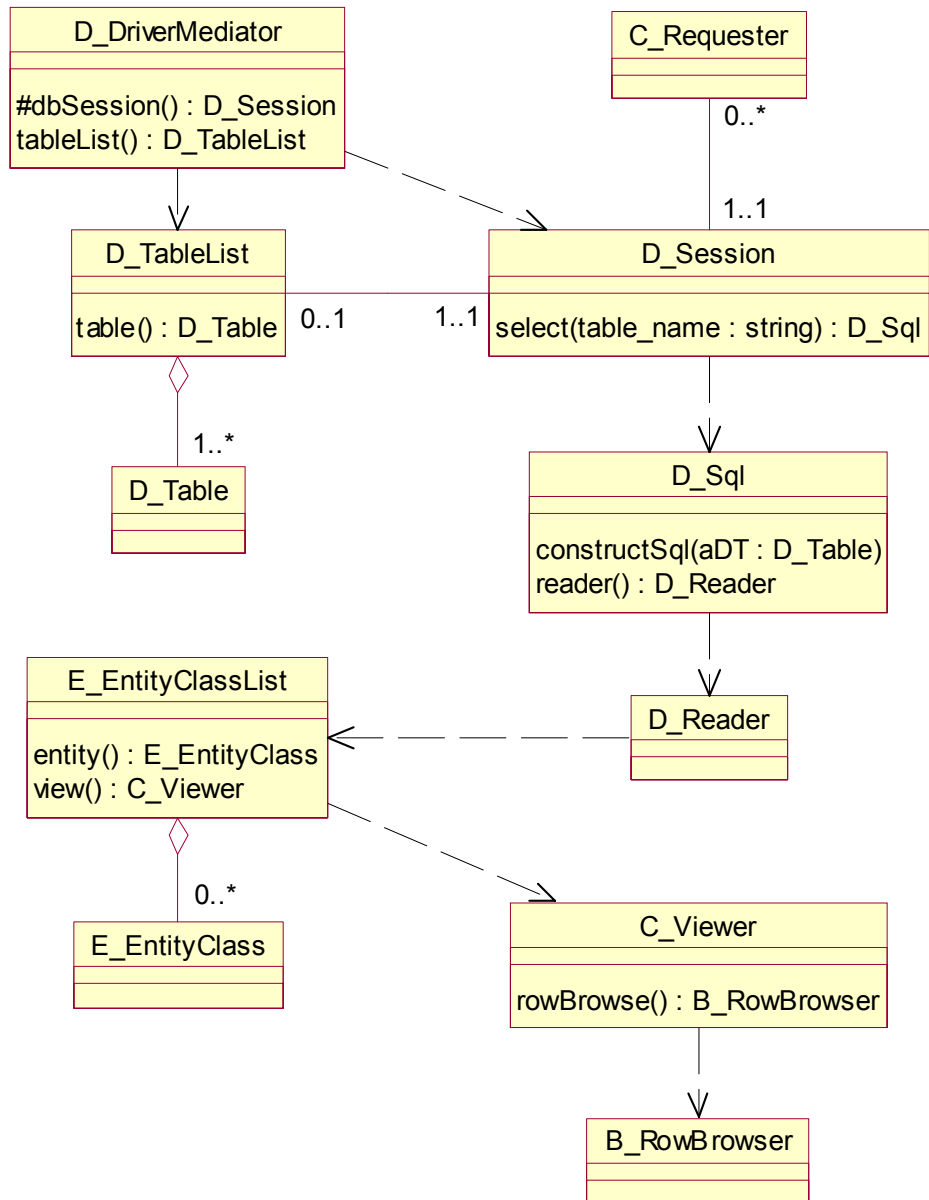
Each entity object `:E_EntityClass` will have corresponding `:D_Reader` and `:D_Updater` objects able to interact with the database. It will also have a corresponding `:C_Viewer` object to display its state in a GUI window. For example, the class `E_Student` will have its own specialized versions of `D_Reader`, `D_Updater`, and `D_Viewer` classes.

9.1.3.1 Structural Collaboration for Persistency

The design of persistency must include the structural and behavioral collaboration models (Section 6.2.4). Figure 9.A demonstrates a structural collaboration model for persistency with a relational database.

The model shows the generic classes. In reality, specialized versions of many classes will be used. The model is incomplete. It only represents the classes that collaborate in a task of selecting data from a table and displaying the results in a row browse window. Finally, the model encapsulates access to a relational database.

FIGURE 9.A
Structural
collaboration for
persistence



The model in Figure 9.A uses arrowed interrupted lines to signify *instantiation relationships*. These relationships implement the *Producer/Product paradigm* to create objects (SourcePro, 2001). The paradigm uses objects of one class (`Producer`) to instantiate objects of another class (`Product`). Rather than invoking public constructors, the producers use private constructors. In effect, instantiation relationships implement run-time *association links* between producers and products.

The client program has to establish a connection to a database before any persistent object can be accessed. The connection is normally established for the duration of program's execution. The connection is obtained from `D_DriverMediator` by providing the database address (URL), the user's name and password. `D_DriverMediator` establishes the connection by loading an appropriate driver from the set of registered ODBC/JDBC or native DB drivers. `D_DriverMediator` instantiates `D_Session` and `D_TableList`.

`C_Requester` represents a client event that requests an interaction with a persistent database object. The association with `D_Session` ensures that all database requests can use single database connection (this minimizes the overhead associated with opening and closing many connections).

`D_Session` instantiates a `:D_Sql` object, which in turn constructs an SQL query string to select data from the table (as supplied in the actual argument to `constructSQL` method). In general `D_Sql` encapsulates all SQL statements.

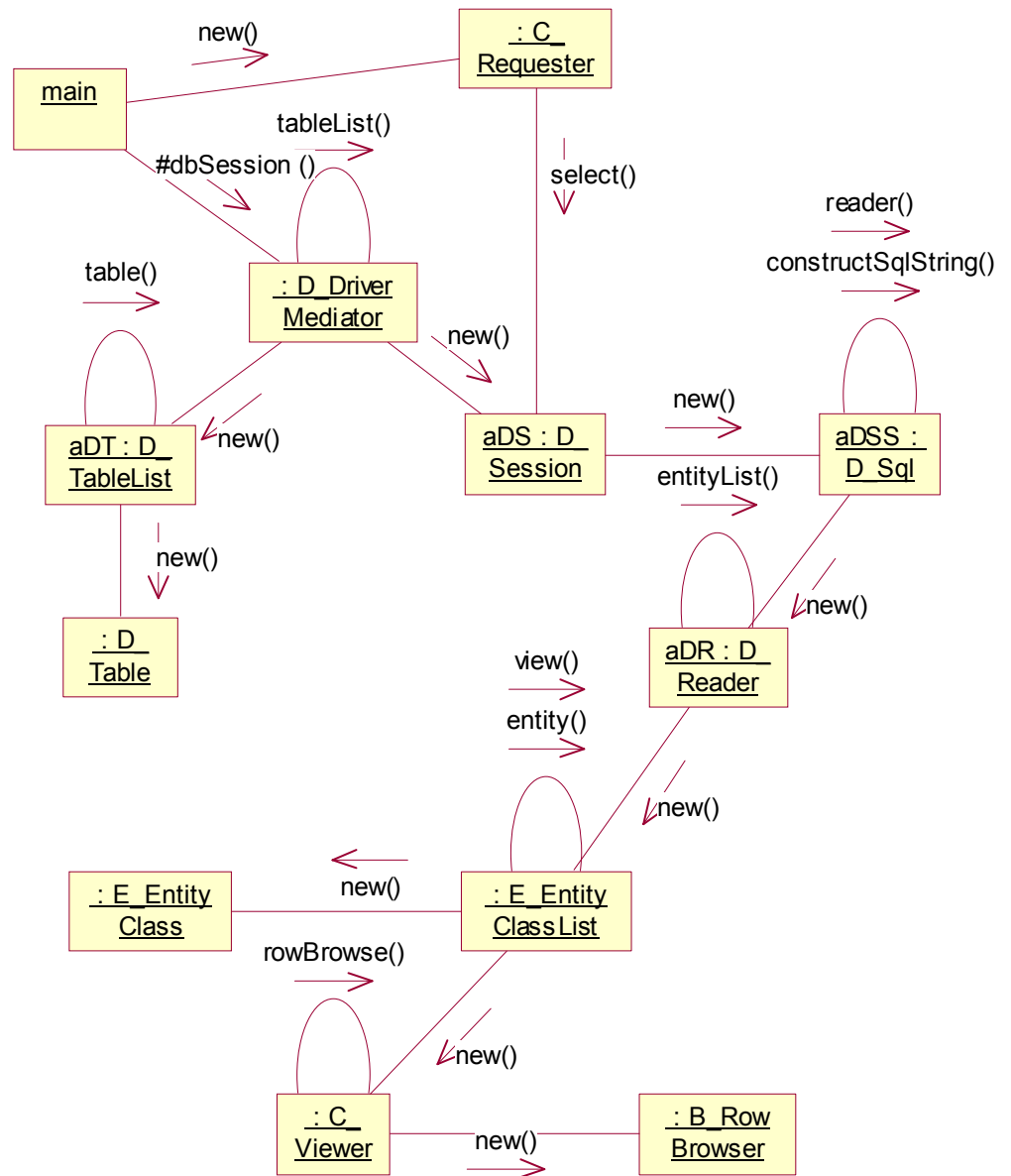
`D_Reader` reads result sets from SQL queries (and results of data manipulation and other SQL statements). For the selection of rows from a single database table, `D_Reader` will instantiate an `E_EntityClassList`. This is a parameterized class that holds instances of `E_EntityClass` (i.e. instances of a specialized version of `E_EntityClass`, such as `E_Student`).

`C_Viewer` is responsible for intercepting entity objects and imitating their display in a GUI window. `B_RowBrowser` manages the display.

9.1.3.2 Behavioral Collaboration for Establishing Database Session

The behavioral collaboration model for selecting data from a table and displaying the results in a row browse window is shown in Figure 9.B. The explanation of classes in the previous Section makes the model self-explanatory. The `new ()` messages signify the *Producer/Product paradigm*. The object labeled `main` signifies the main program.

FIGURE 9.B
Behavioral
collaboration for
persistence



Chapter
10

Testing and Change Management

Bibliography

SourceProDB (2001): *SourcePro DB. Offering Power and Productivity for C++ Database Applications*, White Paper, Rogue Wave Software (accessed from www.roguewave.com on 20-May-2001)