

RESEARCH OUTPUTS / RÉSULTATS DE RECHERCHE

Strategies for Data Reengineering

Henrard, Jean; Hick, Jean-Marc; Hainaut, Jean-Luc

Publication date:
2003

[Link to publication](#)

Citation for published version (HARVARD):
Henrard, J, Hick, J-M & Hainaut, J-L 2003, Strategies for Data Reengineering..

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.



LIBD

Strategies for Data Reengineering

Jean Henrard, Jean-Marc Hick,
Philippe Thiran, Jean-Luc Hainaut

FNRS contact day on «Software (re-)engineering »

Louvain-la-Neuve, 22 may 2003

Laboratory of Database Application Engineering

Laboratoire d'Ingénierie des Bases de Données

www.info.fundp.ac.be/libd



LIBD

Plan

- Introduction
- Problem statement
- Reengineering strategies
 - 2 dimensions
 - 3 strategies
- Conclusion



LIBD

Introduction

- Legacy system =
 - large and old programs build around legacy DBMS
 - vital to the organization
 - significantly resists modifications and changes
 - expensive to maintain
- Solution : migrate to new platform and technologies
 - expensive and complex process
- Incremental strategy is less risky
 - migrate the DB is one of the steps



LIBD

Problem statement

- Data reengineering =
deriving a new database from a legacy database and adapting the software components
 - the functionalities of the system do not change
- Three main steps:
 - schema conversion
 - data conversion
 - program modification



LIBD

Problem statement

- Schema conversion
 - translation of the legacy schema into equivalent schema in the new technology
 - DBRE + database design
- Data conversion
 - migration of the data instances from the legacy system to the new one
 - depends on the schema conversion



LIBD

Problem statement

- Program modification
 - modification of the programs so that they access the new DB instead of the legacy one
 - functionalities, programming language, user interface unchanged
 - complex process that relies on the schema conversion



LIBD

Reengineering strategies

- 2 dimensions
 - *database* dimension (schema migration)
 - *program* dimension (program modification)
- Data conversion is directly dependent on the database dimension

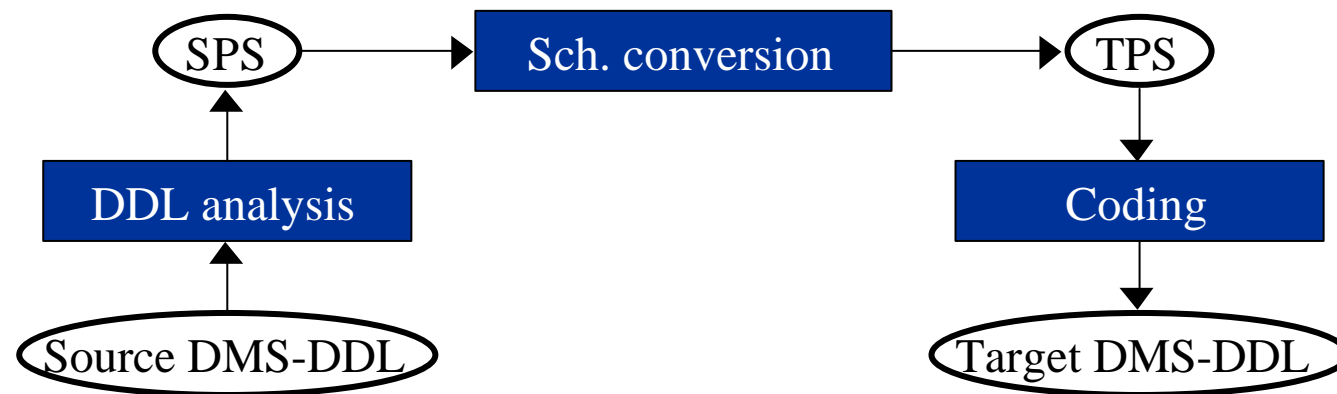


LIBD

Database migration strategies

Physical conversion (D1):

- translate to the closer construct into the target DMS (e.g. 1 file \Rightarrow 1 table)
- no semantic interpretation
- cheap but poor quality DB



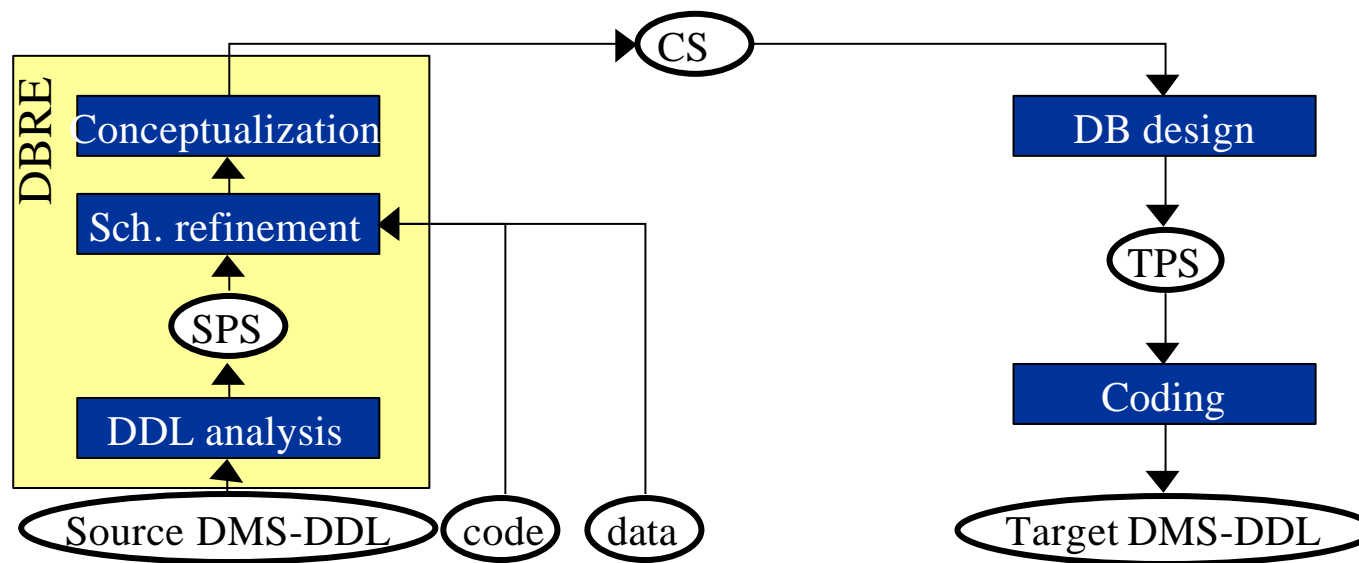


LIBD

Database migration strategies

■ *Conceptual conversion (D2)*

- recovering the semantic (conceptual sch) - DBRE
- developing the new DB from the conceptual sch
- good quality and documented DB but expensive





LIBD

Database migration strategies

- Schema conversion = schema transformation
- History = chain of transformations
- **Mapping** between the source (SPS) and target (TPS) physical schemas
 - = SPS-to-TPS for physical migration
 - = SPS-to-CS-to-TPS for conceptual migration



LIBD

Program modification strategies

■ Wrappers (P1)

- wrappers encapsulate the new database
 - data wrapper =
 - data model conversion
 - semantic conversion
 - functionality simulation
 - “inverse” wrapper: simulate the legacy data interface on the new DB
 - ex: uses COBOL read, write for accessing SQL data
 - SPS -- TPS mapping \Rightarrow automated generation of wrapper
- programs use legacy data access logic
- program logic not changed
- local changes: 1 instruction \Rightarrow x instructions



LIBD

Program modification strategies

■ Statement rewriting (P2)

- legacy DMS-DML \Rightarrow target DMS-DML
ex: replace COBOL file access statement by SQL statement
- rewriting the access statements (new DMS-DML)
 - each legacy DML statement must be located and replaced by equivalent statements in the new DML
 - SPT--PTS mapping \Rightarrow automatic program modification
- program logic not changed
- local changes: 1 instruction \Rightarrow x instructions



LIBD

Program modification strategies

■ Logic rewriting (P3)

- program rewritten to use the new DMS-DML power
 - explicitly accesses new data
 - takes advantage of the new DML
- logic of the program is changed
 - requires a deep understanding of the program
- global change: x instructions \Rightarrow y instructions)

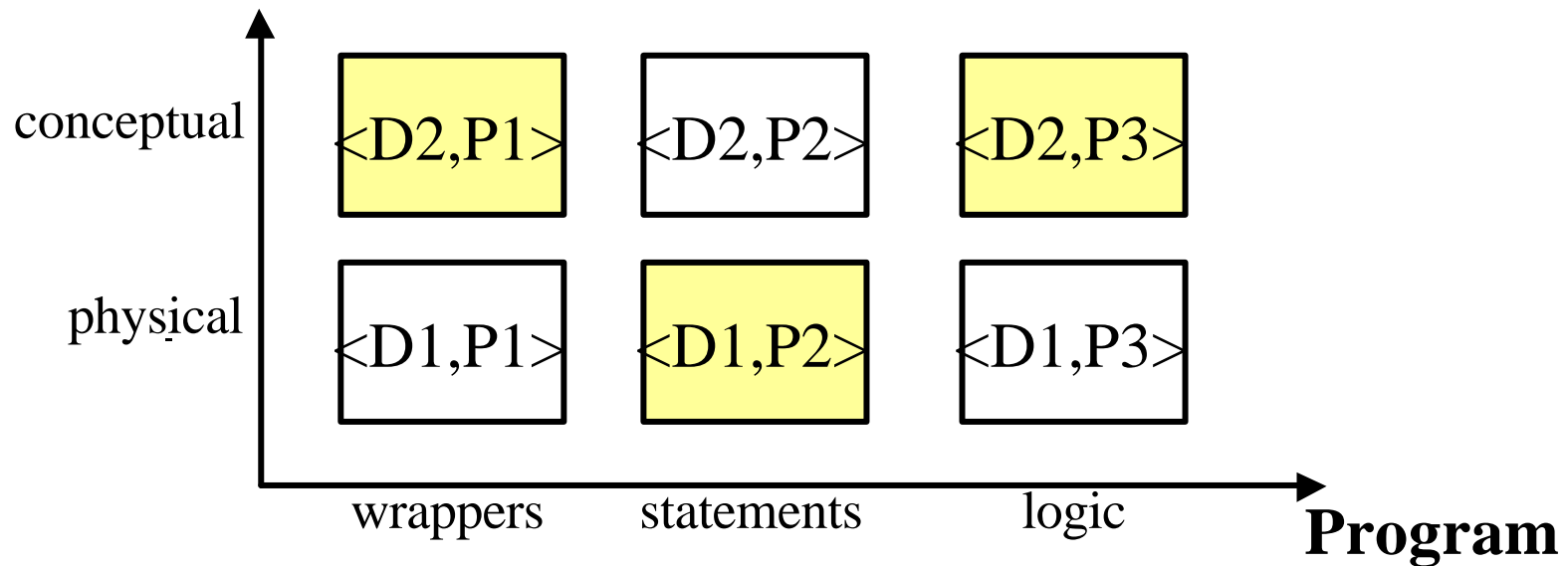


LIBD

Reengineering strategies (summary)

■ Six strategies

Database (schema)



■ $\langle D1, P3 \rangle$ useless



LIBD

Wrapper strategy <D2,P1>

- modification of the legacy code is minimal, new DB well structured and optimized w.r.t. the new DMS
- good solution for complete migration: first the DB and later the programs... Illustration

READ PRODUCT

```
KEY IS PROD-CODE  
INVALID KEY  
    GO TO ERR-123.
```

DELETE PRODUCT

```
END-DELETE.
```



```
CALL WR-ORD-MNGMT  
    USING "READKEY", "PRODUCT",  
        "PROD-CODE",  
        PRODUCT, WR-STATE.  
IF STATUS OF WR-STATE NOT= 0  
    GO TO ERR-123.
```

```
CALL WR-ORD-MNGMT  
    USING "DELETE ", "PRODUCT",  
        " ", PRODUCT, WR-STATE.
```




LIBD

Statement rewriting strategy <D1,P2>

■ Illustration

```
MOVE CUS-CODE TO ORD-CUSTOMER.  
START ORDER KEY >= ORD-CUSTOMER.  
MOVE 0 TO END-FILE.  
PERFORM READ-ORD UNTIL END-FILE = 1.  
READ-ORD SECTION.  
BEG-ORD.  
READ ORDER NEXT  
  
AT END MOVE 1 TO END-FILE  
GO TO EXIT-ORD.  
<<processing current ORD record>>  
EXIT-ORD.  
EXIT.
```



```
EXEC SQL declare cursor ORD_GE_K1 for  
select ORD_CODE,ORD_CUSTOMER,ORD_DETAIL  
from ORDER where ORD_CODE >= :ORD-CODE  
order by ORD_CODE END-EXEC.
```

```
EXEC SQL declare cursor ORD_GE_K2 for  
select ORD_CODE,ORD_CUSTOMER,ORD_DETAIL  
from ORDER where ORD_CUSTOMER >= :ORD-CUSTOMER  
ORDER BY ORD_CUSTOMER END-EXEC.
```

```
...  
MOVE CUS-CODE TO ORD-CUSTOMER.  
EXEC SQL open ORD_GE_K2 END-EXEC.  
MOVE "ORD_GE_K2" to ORD-SEQ.
```

```
IF ORD-SEQ = "ORD_GE_K1"  
EXEC SQL fetch ORD_GE_K1 into :ORD-CODE,  
:ORD-CUSTOMER,:ORD-DETAIL END-EXEC  
ELSE IF ORD-SEQ = "ORD_GE_K2"  
EXEC SQL fetch ORD_GE_K2 into :ORD-CODE,  
:ORD-CUSTOMER,:ORD-DETAIL END-EXEC  
ELSE IF ...  
END-IF.
```

```
IF SQLCODE NOT = 0  
MOVE 1 TO END-FILE GO TO EXIT-ORD.
```

```
<<processing current ORD record>>
```



LIBD

Statement rewriting strategy <D1,P2>

- modification of the legacy code is minimal, DB not restructured, mimics the legacy DB
- Quick and dirty solution....



LIBD

Logic rewriting strategy <D2,P3>

■ Illustration

DISP-ORD.

```
READ ORDER KEY IS ORD-CODE
```

```
  INVALID KEY
```

```
    GO TO ERR-ORD-NOT-FOUND.
```

```
  PERFORM DISP-ORD-CUS-NAME.
```

...

DISP-ORD-CUS-NAME.

```
MOVE ORD-CUSTOMER TO CUS-CODE
```

```
READ CUSTOMER
```

```
  INVALID KEY
```

```
    DISPLAY "ERROR: UNKOWN CUST"
```

```
  NOT INVALID KEY
```

```
    DISPLAY "ORD-CODE: "
```

```
      ORD-CODE NAME.
```



DISP-ORD.

```
EXEC SQL
```

```
  SELECT O.CODE, C.NAME
```

```
  INTO :ORD-CODE, :NAME
```

```
  FROM ORDER O, CUSTOMER C
```

```
  WHERE O.CUS_CODE = C.CODE
```

```
        AND O.CODE = :ORD-CODE
```

```
END-EXEC.
```

```
IF SQLCODE = 0
```

```
  DISPLAY "ORD-CODE:  »
```

```
    ORD-CODE NAME
```

```
ELSE
```

```
  GO TO ERR-ORD-NOT-FOUND.
```



LIBD

Logic rewriting strategy <D2,P3>

- program is rewritten (long, difficult, risky)
new DB well structured and optimized w.r.t. the new DMS
programs optimized w.r.t. the new DMS
- good solution if no program migration planned, only the DB is migrated



LIBD

Conclusion

Strategy	Database migration	Program conversion	Quality
<i>D2, P1</i>	complete DBRE, expensive	cheap, fully automated, wrapper semi- automatically generated	good quality DB, the programs unchanged (call to the wrapper)
<i>D1, P2</i>	cheap, fully automated	cheap, fully automated	poor quality DB, the programs unchanged (call the new DML)
<i>D2, P3</i>	complete DBRE, expensive	very expensive, requires a deep understanding of the programs	good quality DB, programs semi- renovated



LIBD

Conclusion

Strategy	Performance	Maintenance	Evolution
<i>D2, P1</i>	Poor: legacy logic, mismatch, emulation	like the legacy system, but the semantics of the DB is known and data access simulated by the wrapper	easier, the new functions can directly access to the new DB
<i>D1, P2</i>	Poor: legacy logic, mismatch	like the legacy system, the semantics of the DB is not recovered but data access are simulated by the new DML	difficult, the DB simulates the legacy one
<i>D2, P3</i>	Good: new logic, matching	easier, the semantics of the DB is known	easier, the new functions can directly access to the new DB