Sample Design

to explain the Specification Card Approach for

Conceptual Software Design:

WissDB

A system

to store & retrieve Knowledge Items

Part 1:

The Data Model

URI = D_ (WissDB / Design / The Data Model)

This page is empty

Contents

This page is empty

Purpose of this Document

This paper might be part of a series of papers which constitute the conceptual and logical design of the WissDB Archive System which is

- to structure, store and index software engineering knowledge as well as results, such as best practices, sample design or black boxes containing reusable code
- and support the user in finding and retrieving such knowledge in a sufficiently selective, **activity, role, or association related** way.

Associations in this sense are binary associations of different, freely configurable semantics.

Basis of the Design are:

- D_(WissDB / Requirements)

This Document's URI is:

- D_(WissDB / Design / The Data Model)

Document Status

Revision	0.1
Last Update	06/08/2016
Author	Gebhard Greiter
Purpose	This document is to serve as a not too simple example how to create software design in form of Specification Cards, and how to present them in a Project Web (i.e. in HTML, well indexed and heavily hyper-linked across arbitrarily many documents).

This page is empty

Management Summary

This document is to specify a

Conceptual and Physical Data Model

for a system **WissDB** to manage reusable software engineering results and best practices. WissDB is

- to structure, store and index knowledge, design and black boxes containing code
- and support the user in finding and retrieving these items in a sufficiently selective **activity, role, or association related** way.

WissDB, as a concept, consists of

- a data model (= C_WissDB_DM specified in this document)
- an application API (= C_WissDB_API)
- and a dedicated high level storage API (= C_WissDB_DL_API)

DL_API is to be understood as an abstract DBMS with an API supporting in a best possibly way the implementation of WissDB Business Transactions.

API – is the set of all methods that can be directly invoked by WissDB applications. Each method call is designed to the effect that it could be wrapped as a Web Service.

Consequences of this design are:

- WissDB can have presentation layers of any form, especially a web-based user interface easy to integrate into any company's intranet.
- Feeding knowledge (e.g. updated versions of practice instances) to WissDB is easy to automate.
- Extracting knowledge from WissDB in an accountable, easily reproducable way is possible.

To make the use of code generators possible, the data model is specified in a notation that is both easy to parse and easy to read by humans.

The notation we use is specified in the section 5 at the end if this paper (the reader is asked to read it in parallel with section 2).

Together with this document you should have received a HTML presentation of the WissDB Entity Relationship Diagram (a file **WissDB.ERD.htm** easier to maintain and therefore to be used as the final reference – less important attributes may be described only there).

1 WissDB: Data Model & Data Model Semantics

This section describes

C_ WissDB_DM : The WissDB data model from a user's point of view.

We specify it in ERD Notation (a formal language described in the last chapter of this document). The physical data model is described in section 3 via SQL CREATE TABLE statements.

Pre-considerations

One important requirement on the data model we need is that it must not require Result instances to have any specific structure. What me mean is: Though we need some structure, this structure should not be more than a WissDB-specific view the user should have not problems to define and to use.

Our solution idea is to have this view present in form of a hierarchical structure that is given by logical locations (unique resource identifiers that are not simply atomic names but have a structure to let us see how items are nested into each other in the view of WissDB). The domain representing these identifiers is **D_ Locator**:

- d **D_ Name** VARCHAR(80)

  Values of type D_Name are case-insentive. Each of them must be a string that could be used as a name for a file in Microsoft's NTFS file system.

- d **D_ Locator** VARCHAR(255)

  A value of type D_Locator is a string N/ or x/N/ such that N is a D_Name, and x/ is either empty or again a D_Locator (x could be a number which is then called a **Project Locator Alias**).

  A D_Locator is called a **Schema Locator** if and only if the last name N in the locator is any of the following:

  . Process
  . Role
  . Result
  . Description
  . Structure
  . Aspect

Locators have to be seen as logical URLs (so-called URIs). The WissDB server is to map them to concrete URLs (resp. onto resources that are to be protected by access permissions).

As we will learn in the following, **Instance Locators** do always start with a corresponding schema locator.

Example: This document here has for its instance locator the URI WissDB/Result/WissDB/Design/Data Model Specification (the last slash in locators is seen only in the view of the DBMS).

It is important to note that WissDB is managing, first of all, meta data representing information about knowledge items. To store knowledge items itself, WissDB may rely on one or more other systems.

Before we now start to design entity types, let us define all domain types needed. Please note that that the boxes Item Type and Description Type shown in the picture above collapse into only one domain type D_ItemType: The Description of a Result will from now on be seen as being a specific part of the Result. It should be given a locator matching the pattern

D_Locator / **Result** / Result Name / **Description**.

Example: If in the WissDB Project we had a document specifying what the outcome of the design phase should be, this paper's URI would be `WissDB/Result/Design/Description`.

Note: `WissDB/Result/Design` is to be understood as a result type (not as an activity). An activitity may have results of different types.

In addition to D_Locator, subsystem WissDB of WissDB defines the following domain types:

- d **D_ ItemType** INT

Valid values are:

- v . Description of Process
- v . Description of Role
- v . Description of Result
- v . Description of Practice Candidate
- v . Practice Candidate (a zipped Knowledge Package)
- v . Solution Requirement
- v . Solution Concept
- v . Solution Code
- v . Business
- v . Technology
- v . Advice
- v . Information

Please note: In this and also all the following domain types the set of valid values should be capable of being redefined when need arises. So, what we show in this document, is more or less a suggestion only.

To give an example: In the picture on page 3 there is a value **Solution** mentioned. In the design here we have split it up into three more specific values (**Solution Requirement**, **Solution Design**, and **Solution Code**).

To implement WissDB in a way guaranteeing such flexibility should be seen as an important requirement.

- d **D_ PracticeType** INT

Valid values are:

- v . Best Practice
- v . Lesson Learned

- d **D_ ViewType** INT

Valid values are:

- v . Concept
- v . Implementation

```
- d          D_ AbstractionType      INT

            Valid values are:

- v          .  Solution
- v          .  Template
- v          .  Pattern
- v          .  Strategy


- d          D_ UsageType            INT

            Valid values are:

- v          .  Sample
- v          .  Reference
- v          .  Use after customization
- v          .  Use as is


- d          D_ CorrelationType      INT

            Valid values are at least:

- v          .  B is Solution Concept for Requirement A
- v          .  B is Code implementing Concept A
- v          .  B is Solution Concept based on Technology A

            During the lifetime of WissDB many more such values might
            be added.
```

In order to ensure that the set of valid values for enumeration domain types can be reconfigured (or at least extended), we implement an auxiliary table documenting these values:

```
- ec         E_ DomainValues

- eca,pk     A_DomName               D_DomainName
- eca,pk     A_DomValue              D_ValueName
- eca,nn     A_ValueAsNr             D_ValueNumber
- eca        A_Semantics             D_Comment
- eca        A_ObsoleteSince         D_Date
```

Having defined all domain types needed, we are now ready to define the WissDB entity types:
There are four core entity types in WissDB. They model **Processes, Activities, Roles,** and **Knowledge Items**:

```
- ec            E_ Process

                |
- eca,pk        A_Loc                       D_Locator
- eca,nn        A_Description               E_KnowledgeItem
- eca           A_Role                      E_Role

                Each process locator is to match the pattern
                D_Locator/Process/Process_Locator (where the Process_Locator
                may contain slashes: A process is seen as an activity that is
                broken down hierarchically in subprocesses which, depending
                on the concrete contex, you may see as processes, phases or
                simple atomic tasks).


- ec            E_ Role
                |
- eca,pk        A_Loc                       D_Locator
- eca,nn        A_Description               E_KnowledgeItem


- ec            E_ KnowledgeItem
                |
- eca,pk        A_Loc                       D_Locator
- eca,nn        A_Type                      D_ItemType
- eca           A_NodeValue                 D_NodeValue

                Containment of Knowledge Items is reflected via the D_Locator
                values in A_Loc (which are the knowledge items' URIs).
```

Type **E_KnowledgeItem** has specializations. They model **Practice Candidates**, **Results** and (accepted) **Practices**. Furthermore we have, on the set of all knowledge items, a generic relation **R_Is_related_to**. It is to allow us to model binary item associations of different semantics:

```
- ec            E_ Candidate
                |
- eca,pk        A_                      e_KnowledgeItem
- eca,nn        A_from                  D_EMailAddress
```

```
- ec            E_ Result
                |
- eca,pk        A_                      e_KnowledgeItem
- eca,nn        A_View                  D_ViewType
- eca,nn        A_Abstraction           D_AbstractionType
- eca,nn        A_Usage                 D_UsageType
- eca           A_isResultOf            E_Process

                Valid values of type E_Result.A_Loc need to match the
                pattern D_Locator/Result/D_Name.

                All items that are seen as part of such a Result have to have
                a locator being prefixed by the Result's locator.
```

```
- ec            R_ Is_related_to
                |
- eca,pk        A_A                     E_KnowledgeItem
- eca,pk        A_B                     E_KnowledgeItem
- eca,pk        A_Correlation           D_CorrelationType
```

This model implies that each result may be a hierarchy of smaller knowledge items. The nesting is given via the item locators. Furthermore – because of the generic relation R_Is_related_to – a result can also have correlation structure.

```
- ec            E_ Practice
                |
- eca,pk        A_                      e_Result
- eca           A_PracticeType          D_PracticeType
- eca           A_reuse_0               D_Counter
- eca           A_reuse_1               D_Counter
- eca           A_reuse_2               D_Counter
- eca           A_reuse_3               D_Counter
```

```
Semantics:

A_reuse_0 = number of downloads
A_reuse_1 = number of ratings "reuse value minimal"
A_reuse_2 = number of ratings "reuse value moderate"
A_reuse_3 = number of ratings "reuse value quite high"
```

We see: Specific results can be marked to be either **Best Practice**, or **Lesson Learned**.

Users who downloaded such a **Practice** instance could some time afterwards receive an e-mail asking them for a rating. The data model allows to maintain such rating results.

Note also: If a result is a practice instance, it may loose this quality later on (because it is always possible that better practices are found, or simply because technology changes to the effect that previous best practice solutions are no longer acceptable).

Given the fact that only Results can be Practice items, two questions could be asked:

- Could it be useful also to support the classification of any item as Best Practice or Lesson Learned?
- Could it be useful to support practice instances that are a set of results?

The current design does not support practice instances to be a sequence of results that are not nested into each other

- because that would complicate the model,
- because practice instances should not get too large anyway (the smaller a practice instance is the greater the chance that it will be reused), and
- because via their locators you always could give results a hierarchical structure: a structure nesting results into more complex results.

There could, e.g. be a result

The WissDB System

and nested therein

The WissDB System/ **Result**/ The WissDB System.

You should also note that the data model proposed here does not force us to assign, to a given result, a specific process – we are only allowed to do so.

Though we do not support classifying each item as Best Practice or Lesson Learned, the user will always be able to do so by choosing a Locator that makes that item a result.

So you see: A **Result** is a knowledge item we can associate with an **Process** (i.e. a named activity). Having done so, the result may also be associated with a **Role**. We can – but need not – make it a **Practice Instance**.

Results should always be well documented, and so there could e.g. be a convention saying that each Result is to have a **Description** (our data model does not enforce this per se, but item locators will always show you whether there is such documentation: Items that are result descriptions are to have a locator matching the pattern

D_Locator/ **Result**/ NameOfResult/ **Description**

The last part of the WissDB data model is to support the indexing of knowledge items:

```
-  ec            R_ Is_keyword_for
                 |
-  eca,pk        A_keyword                  E_Aspect
-  eca,pk        A_for                      E_KnowledgeItem


-  ec            E_ Aspect
                 |
-  eca,pk        A_Loc                      D_Locator

                 If X/Y is an E_Aspect.A_Loc, then X is said to be a
                 Knowledge Area for which Aspect Y makes sense.

                 Examples could be:

                     X = Software/Implementation
                     Y = Technology
                 or
                     X = Project Management
                     Y = Risk Management/Checklist
```

The rationale for this modeling is: If a user is indexing an item by associating keywords to it, he should be asked to do so by first selecting a knowledge area and then, in a second step, one or more existing aspects (which again could be knowledge areas).

To have such a structure on the set of all keywords allowed will help us to restrict any search for result items to quite specific knowledge areas.

Finally we have means to associate processes and results to concrete projects. This however is an additional view applications may or may not have use for:

```
- d            D_ ProjectLocator   Positive Integer

- ec           E_ Alias
                |
- eca,pk       A_Nr                        D_ProjectLocator
- eca          A_Loc                       D_Locator

               Values of type E_Alias.A_Loc are not allowed to contain one of
               the reserved names Aspect, Process, Role, Result, Description, Structure,
               or Selector. They may however start with a D_ProjectLocator
```

How to work with Aspects

Knowledge areas could be, e.g.

- Area **Project Management** with aspects
  - Time Management
  - Cost Management
  - Quality Management
  - Team Management
  - Risk Management
- Area **Software Development** with aspects
  - Analysis
  - Requirements Management
  - Design
  - Implementation
  - Test
  - Delivery
  - Support

The keyword **Prototyping** could make sense in the context of **Risk Management** and also in the context of **Implementation**, and so

Aspect/ Project Management/ Risk Management and
Aspect/ Software Development/ Implementation

should both be knowledge areas containing **Prototyping** as an aspect (or even a subarea).

If the user would then search for knowledge via a query

Aspect = Project Management/ Risk Management/ Prototyping,

only items would be found that speak about prototyping in the context of risk management.

To have, in this sense, **keywords in context** (not just keywords) is very helpful and should be considered an important requirement.

Knowledge Packages

Knowledge that shall be imported into WissDB as well as knowledge that is to be exported (as a search result) is exchanged between user and system in form of knowledge packages:

A **Knowledge Package** is a zipped tree of files representing

- process structure,
- knowledge items,
- attributes of knowledge items,
- and also correlation structure.

A knowledge package is said to be **well formed** if:

- For each file in the package the path starting with the package root and ending with the name of the file is a D_Locator.
- Directly under the root of the package there is a file named **Structure**.
- Directly under each subtree root named **Result**/ there is also **Structure** file.
- All paths starting under the root of the package start with a Schema Locator.
- Each **Structure** file is ASCII text in Knowledge Structure Format describing all nodes found in the tree that is rooted in the node of which this file is a son (structure files ignored).

A file containing ASCII text is said to be in **Knowledge Structure Format** if:

- The first column of each line is ASCII character 32 or 45 (a space or a minus sign).
- The second column of each line is ASCII character 32 (a space)
- If the first column contains a minus sign, the string starting in column 3 is a D_Locator (relative to the node under which the structure file is found).
- Directly following such a line may be lines starting with ASCII characters 32, 32, 46, 32, 32 followed by a string **X: Z** so that **X** is denoting an attribute and **Z** a value for this attribute. (ASCII character 46 is the dot).
- Please note that an attribute **X** in this sense can also be a correlation type (or the name of the relation **Is_keyword_for**).

- If the value **Z** is a D_Locator not starting with a number, it must be given relative to the node under which the structure file is found. It must have this form if X is a value of D_CorrelationType. This is to ensure that knowledge packages and items therein that are results – or even practice instances – will always be self contained.

Rationale for the Knowledge Structure Format:

The reader may wonder why we do not require structure files to be in XML format. The reason for this decision is that knowledge administrators – and especially people submitting results to be included into the knowledge database – shall be able to read and edit structure files in a painless way.

Note also that structure files may contain comment (comment are all text sections not starting with a line containing a minus sign in their first column). Comment sections should always follow an empty line.

Rationale for the Format of Knowledge Packages:

As long as a knowledge package is not zipped, it is simply a tree of files (i.e. a data structure the user and knowledge administrator is used to work with). This will also minimize the need for creating values of type D_Locator explicitly.

Dialogs to be supported by WissDB can be quite simple, and structure files can be generated to a very large degree by a suitable utility that is capable of being envoked via e.g. ANT, make, or nmake.

2 WissDB: The Logical Data Model

Because the preceeding section did not allow any redundancy in the specification, we now show the result in terms of a complete Entity Relationship Model.

Notation semantics are explained at the end of this diagram (the diagram is given in form of text derived automatically from the formal data model specification in section 2. It is object oriented in as far as the description of an entity type is always embedded into a section showing in detail also all its super- and subtypes.

The description of an entitiy type includes all structure of that type, i.e. attributes and relationships).

c: **E_DomainValues**

.pk	D_DomainName	a_DomName
.pk	D_ValueName	a_DomValue
.nn	D_ValueNumber	a_ValueAsNr
.	D_Comment	a_Semantics
.	D_Date	a_ObsoleteSince

c: **E_Process**

.pk	D_Locator	a_Loc
.nn	E_KnowledgeItem	a_Description
.	E_Role	a_Role

<-- can occour as: E_Result.of

c: **E_Role**

.pk	D_Locator	a_Loc
.nn	E_KnowledgeItem	a_Description

<-- can occour as: E_Process.Role

c: **E_KnowledgeItem**

```
.pk  D_Locator              a_Loc
.nn  D_ItemType             a_Type
.    BLOB                   a_NodeValue

<-- can occour as:      R_Is_keyword_for.for
<-- can occour as:      R_Is_related_to.B
<-- can occour as:      R_Is_related_to.A
<-- can occour as:      E_Role.Description
<-- can occour as:      E_Process.Description
```

```
|
```

E_Result

```
|    e_KnowledgeItem
.nn  D_ViewType             a_View
.nn  D_AbstractionType      a_Abstraction
.nn  D_UsageType            a_Usage
.    E_Process              a_of
```

```
|
```

E_Practice

```
|    e_KnowledgeItem
|    e_Result
.    INT                    a_PracticeType
.    D_Counter              a_reuse_0
.    D_Counter              a_reuse_1
.    D_Counter              a_reuse_2
.    D_Counter              a_reuse_3
```

```
|
```

E_Candidate

```
|    e_KnowledgeItem
.nn  D_EMailAddress         a_from
```

c: **R_Is_related_to**

```
.pk  E_KnowledgeItem        a_A
.pk  E_KnowledgeItem        a_B
.pk  D_CorrelationType      a_Correlation
```

c: **R_Is_keyword_for**

```
.pk  E_Aspect               a_keyword
.pk  E_KnowledgeItem        a_for
```

c: **E_Aspect**

```
.pk  D_Locator              a_Loc

<--  can occur as:          R_Is_keyword_for.keyword
```

c: **E_Alias**

```
.pk  D_ProjectLocator       a_Nr
.    D_Locator              a_Loc
```

Notation Semantics:

Prefixes E_, R_, S_, A_, e_ in front of a name show what kind of concept the name x in question is denoting:

E_x is an entity class. Note however:

We write e_x instead of E_x where we want to say that E_x is occurring in the role of a superclass.

If a class E_x1 is a specialization of another class E_x, i.e. if each instance of E_x1 can be also be seen as an instance of type E_x, then e_x is meant to be the group of all attributes of E_x1 defined in E_x already (the set of all inherited attributes).

R_x is entity class E_x representing entity associations that are not of cardinality 1:n or n:1 (they need not even be binary relations).

A_x is an attribute of an entity class, same as a_x() in C++ or Java.

S_x is a complex attribute (i.e. a set of attributes), same as a_x() in C++ or Java.

We write a_x where we do not care to see whether attribute x is complex or not.

D_x is a domain, i.e. an attribute type.

pk says that this attribute is part of the entity's primary key.

nn says that this attribute must be NOT NULL.

c: is to mark core entity types (types that do not specialize any other type). Types not marked as core entity types inherit structure from their - always unique - direct supertype).

A Core Type is an entity type that is not a specialization of another entity type in the model shown.

Specialisation hierarchies are shown in this form:

```
|
E_x3
     |    e_x1
     |    e_x2
     .    E_x3 structure not inherited
```

means: E_x3 spezializes E_x2 (so that all structure of E_x2 is also structure of E_x3). E_x3 is a direct subtype of E_x2, and E_x2 is a direct subtype of E_x1.

Entity Relationship Structure is shown as follows:

E_x1

```
      ->>  set of E_x2   a_Name
```

means: Given any object x1 of type E_x1, x1.a_Name() is an instance of a List h_Name defined in the ERD (the Name can be seen as describing the role of the list members in relation to x1).

E_x1

```
      .    E_x2  x3      (to be read as:   <-- E_x2.x3   )
```

means: Each value E_x1.A_x3 is primary key of an entity of type E_x2, so that (E_x1, E_x2) is a relation of cardinality (m:1) or (1:1).

E_x1

```
      .    D_x2  x3
      .    S_x2  x3
```

means: Each value E_x1.A_x3 is an attribute of type D_x2 (resp. S_x2, S_x2 a set of attributes such that records of type S_x2 make sense in themselves).

R_x1 is a type E_x1 that can be interpreted as a relationship type of a specific dimension n, 2 <= n.

Note: Classes E_x without relation structure --> or <<- tend to be superflous. They form a self-contained ERD model which quite often does not make sense in isolation.

Here is an example: Classes meant to be enumerations such as e.g.

E_Category

```
      .   A_Value           D_ValueName
      .   A_Semantics       D_Comment
      .   A_ObsoleteSince   D_Date
```

need usually not exist in form of a separate table. To model it in form of a domain D_Category together with an attribute A_ObsoleteSince in the table E_DomainValues would be better:

E_DomainValues

```
      .   A_Domain          D_DomainName
      .   A_Value           D_ValueName
      .   A_Semantics       D_Comment
      .   A_ObsoleteSince   D_Date
```

For WissDB it is mandatory to implement the table E_DomainValues describing enumeration types because otherwise their **Values allowed** would not be configurable.

3 WissDB: The Physical Data Model

From the formal specification given in section 2, the following physical data model is derived.

The reader may wonder why there is an additional table **DomainValues**. Its content is meant to document

- which values are actually allowed in a domain that is an enumeration of constants
- and also what these constants mean (semantics).

To have such documentation in the database already is to make its content self-describing. Not to have a table **DomainValues** would be acceptable only if the constants were not coded to be – in their physical presentation – integer values.

```
CREATE TABLE   DomainValues
(
               UpdateNr             INTEGER        NOT NULL
,              DomName              CHAR(20)       NOT NULL
,              DomValue             CHAR(240)      NOT NULL
,              ValueAsNr            INTEGER        NOT NULL
,              Semantics            CHAR(240)
,              ObsoleteSince        DATE
,

PRIMARY KEY    (
               DomName
,              DomValue
               )
)
;
```

```
CREATE TABLE   Aspect
(
               UpdateNr             INTEGER        NOT NULL
,              Loc                  VARCHAR(255)   NOT NULL
,

PRIMARY KEY    (
               Loc
               )
)
;
```

```
CREATE TABLE   Process
(
               UpdateNr             INTEGER        NOT NULL
,              Loc                  VARCHAR(255)   NOT NULL
,              DescriptionLoc       VARCHAR(255)   NOT NULL
,              RoleLoc              VARCHAR(255)
,

PRIMARY KEY    (
               Loc
               )
)
;


CREATE TABLE   Role
(
               UpdateNr             INTEGER        NOT NULL
,              Loc                  VARCHAR(255)   NOT NULL
,              DescriptionLoc       VARCHAR(255)   NOT NULL
,

PRIMARY KEY    (
               Loc
               )
)
;


CREATE TABLE   Is_related_to
(
               UpdateNr             INTEGER        NOT NULL
,              ALoc                 VARCHAR(255)   NOT NULL
,              BLoc                 VARCHAR(255)   NOT NULL
,              Correlation          INTEGER        NOT NULL
,

PRIMARY KEY    (
               ALoc
,              BLoc
,              Correlation
               )
,

CONSTRAINT     fk1_Is_related_to
FOREIGN KEY    (
               ALoc
               )
REFERENCES     KnowledgeItem
               (
               Loc
               )
ON DELETE SET NULL
,

CONSTRAINT     fk2_Is_related_to
FOREIGN KEY    (
               BLoc
               )
REFERENCES     KnowledgeItem
               (
               Loc
               )
ON DELETE SET NULL
)
;
```

```
CREATE TABLE   Is_keyword_for
(
               UpdateNr             INTEGER        NOT NULL
,              keywordLoc           VARCHAR(255)   NOT NULL
,              forLoc               VARCHAR(255)   NOT NULL
,

PRIMARY KEY    (
               keywordLoc
,              forLoc
               )
,

CONSTRAINT     fk1_Is_keyword_for
FOREIGN KEY    (
               keywordLoc
               )
REFERENCES     Aspect
               (
               Loc
               )
ON DELETE SET NULL
,

CONSTRAINT     fk2_Is_keyword_for
FOREIGN KEY    (
               forLoc
               )
REFERENCES     KnowledgeItem
               (
               Loc
               )
ON DELETE SET NULL
)
;


CREATE TABLE   Union_KnowledgeItem
(
               UpdateNr             INTEGER        NOT NULL
,              OfType               CHAR(100)
,              Loc                  VARCHAR(255)   NOT NULL
,              Type                 INTEGER        NOT NULL
,              NodeValue            BLOB
,              View                 INTEGER        NOT NULL
,              Abstraction          INTEGER        NOT NULL
,              Usage                INTEGER        NOT NULL
,              ofLoc                VARCHAR(255)
,              PracticeType         INTEGER
,              reuse_0              INTEGER
,              reuse_1              INTEGER
,              reuse_2              INTEGER
,              reuse_3              INTEGER
,

PRIMARY KEY    (
               Loc
               )
)
;
```

```
CREATE VIEW     KnowledgeItem
AS SELECT *
                UpdateNr
,               Loc
,               Type
,               NodeValue

FROM            Union_KnowledgeItem
WHERE           OfType = Result
OR              OfType = Practice ;
```

```
CREATE VIEW     Result
AS SELECT *
                UpdateNr
,               Loc
,               Type
,               NodeValue
,               View
,               Abstraction
,               Usage
,               ofLoc

FROM            Union_KnowledgeItem
WHERE           OfType = Result
OR              OfType = Practice ;
```

```
CREATE VIEW     Practice
AS SELECT *
                UpdateNr
,               Loc
,               Type
,               NodeValue
,               View
,               Abstraction
,               Usage
,               ofLoc
,               PracticeType
,               reuse_0
,               reuse_1
,               reuse_2
,               reuse_3

FROM            Union_KnowledgeItem
WHERE           OfType = Practice ;
```

```
CREATE VIEW     Candidate
AS SELECT *
                UpdateNr
,               Loc
,               Type
,               NodeValue
,               from

FROM            Union_KnowledgeItem
WHERE           OfType = Candidate ;
```

Note: The attribute **UpdateNr** found in each record is a record-specific version number. It is helpful to detect conflicts in the context of semi-transactions (i.e. non-atomic, so-called long transactions).

Note also: Because the WissDB database must not allow different objects to have the same D_Locator value in their A_Loc attribute, we need to implement an index guaranteeing this restriction database- wide. It must be an index containing normed versions of the D_Locator values because locator values, when used to represent file paths, may not be case-sensitive: They are not case-sensitive under MS Windows, WissDB however is to store them respecting case (as Unix does).

This index should also know the case-sensitive version of each locator (which is defined to be the first version given to the database) and should be used to guarantee, that – if a locator X is a first part of another locator Y, this substring of Y is always shown exactly as X itself is shown.

4 ERD Notation: A formal Language to specify Data Models

ERD Design Notation is a formal language

- **to specify Entity Relationship Data Structures of the most general type**,

i.e. also those in which

- Relationships may have attributes, and
- Relationships may be n-ary (i.e. may not be just binary relationships).

What you describe are:

- Entity types (= entity name, and a sequence of typed entity attributes), and
- Domain types.

A typed attribute in this sense is

- an attribute with values in a specified Domain, or
- an attribute with values in a specified Entity type (generating relationships).

In order to avoid redundancy, you can – for abbreviation –

- use structures (i.e. named sets of domain-valued attributes), and
- use subclassing (i.e. see an entity type as a set of attributes which is a subset of another, more specific entity type)

Whenever you want to refer to a specific domain, entity type, attribute, or set of attributes, you give its ***typed name***, i.e. its name prefixed by

- **D_** for **Domain**
- **E_ e_** for **Entity Type** respectively **Entity Type seen as a Set of Attributes**
- **A_** for **Attribute**
- **S_** for **Set of Attributes**

The code generator takes for input an ASCII file but will see only lines of the following form (if the hyphen is in the first column of the file and is followed by one and only one space):

```
- ec        E_name
- eca       A_           e_name
- eca       A_name       D_name
- eca       A_name       S_name
- d         D_name       Name of an SQL_type supported by your DBMS
```

Instead of

```
- eca
```

you are to write

- **eca,pk**

  if these attribute (or set of attributes) is part of the primary key of the entity in question. You are to write

- **eca,k***n*

  n a positive integer, if the attribute (or set of attributes) is part of a secondary key called k*n*. Because keys (seen as sets of attributes) may overlap, you can have lines such as, e.g.

- **eca,pk,k***2*,k*5*

  If an attribute must not be allowed to have NULL values, say so by saying

- **eca,nn**

  in the line specifying a name for this attribute (or set of attributes).

A small Sample Specification

```
-  ec             E_Book
                  |
-  eca,pk         A_Title             D_Title
-  eca            A_Author            E_Person


-  ec             E_Person
                  |
-  eca,pk         A_FirstName         D_Name
-  eca,pk         A_LastName          D_Name
-  eca,nn         A_Gender            D_Gender


-  d              D_Title             CHAR(80)
-  d              D_Name              CHAR(20)


-  d              D_Gender            INT
                  Valid values are:
-  v              .  male
-  v              .  female
```