Sample Design

to explain the Specification Card Approach for

Conceptual Software Design:

WissDB

A system

to store & retrieve Knowledge Items

Part 2

WissDB seen as an Application Service

D_ (WissDB / Design / The Application Programming Interface)

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 / Purpose and Requirements)
- D_(WissDB / Design / The Data Model)

This Document's URI is:

- D_(WissDB / Design / The Application Programming Interface)

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

WissDB as an Abstract Data Type

i.e. as the set of Methods WissDB Applications are allowed to envoke:

C_WissDB_API

WissDB is a system to manage reusable Software Engineering results and practices.

All parts of the system that have a name starting with **S_** are to be implemented as self-contained methods that, if called, have to have atomic effect on the systems's physical database.

Consequences of specifying WissDB in form of an Abstract Data Type 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, easy to reproduce way is possible.

The document contains essentially two parts:

- Section **Data Views** is a description of all necessary data views, especially of those that represent business objects.

  Each view is described and discussed only in as far as it is not already discussed in D_(Design / WissDB / The Data Model).

  From a user's point of view the most important business objects are

  - **B_Knowledge_Package**
  - **B_Query_Specification**
  - **B_Search_Result**

- Section **Services** describes functionality available via a WissDB user application programming interface on top of which a web-based dialog interface as well as a command line interface could be built.

Notation and Terminology

Each concept specified in this design paper has an identifier starting with a prefix telling you the type of the concept. Prefix semantics are:

- C_… = A WissDB Component
- S_ = A service offered by a WissDB Component in form of at least an API
- m_… = Is so far a manual process only
- P_… = Business Process
- A_… = Actor (a user role or a system role)
- D_… = The logical name of a Document
- V_… = A logical data view on top of the physical data stored in the BOS database. Such a view must be definable via SQL (so that standard reporting tools can be applied).
- B_… = Business Object type (a B_x is a V_x such that all instances of type B_x are owned by a unique WissDB Component. Ownership is also the right to define this view, and so this component is called the Definition Owner).

WissDB Business Transactions

Component:	**C_WissDB_API** part of C_WissDB **API – the system's Application API** – is an abstract specification of the set of services available to WissDB applications and, via C_WissDB_GUI, also to human users who want to use it interactively. C_WissDB_CLI is a command line interface equivalent to C_WissDB_API.
Abstract:	This component offers services to **capture**, **classify**, **correlate** and **communicate** practice instances and sample results in the context of processes for requirements engineering, software development, software support and software maintenance. WissDB can be used to manage reusable software components as well as much more abstract reusable project results.
Has to:	**Interprete or create objects of type:** – B_Knowledge_Package – B_Query_Specification **Implement**, for the user, the following services: – S_Check_Package – S_Save_Package – S_Open_Practice_Candidate – S_Reject_Candidate – S_Accept_Candidate – S_Search_for_Candidates – S_Search_for_Knowledge – S_Export_Descriptions – S_Export_Knowledge – S_Describe_Knowledge_DomainValues – S_Replace_or_create_Knowledge_DomainValues – S_Rename_Project – S_Testsupport_New_Knowledge_Base

	Make all these functions available at least via – the command line interface C_WissDB_CLI, and – a REST conform web-based user interface C_WissDB_via_HTTP. **Each of these functions must be protectable via** C_Access_Control **as a separate resource** (so that, depending on their role, users or applications might be restricted to use only some of these functions). **User Roles to be supported** by C_WissDB_API are – A_Knowledge_Provider – A_Knowledge_User – A_Knowledge_Administrator – A_Knowledge_Schema_Administrator – A_Tester All the services (= functions) named above have to be available, to the end-user, via a graphical, web-based user interface, i.e. via a web browser such as MS Internet Explorer. They also have to be available via a suitable Java or C++ API. A command line interface (CLI) must allow the user to invoke each service S_… also from the NT command window or out of a batch file. The implementation of all these interfaces has to comply with the rules specified in C_WissDB_API, C_WissDB_CLI, and C_WissDB_GUI.
Because of:	Requirements detailed in D_(WissDB / Purpose and Requirements).

Actors and Business Processes supported by C_WissDB_API

A_ System_Administrator

Responsibilities are:

- Install the system.
- Make sure that the system is up and running most of the time during the company's standard working hours.
- Ensure that there are no bottlenecks as a result of large numbers of users working with the system in parallel.
- Ensure that the B_WissDB_Database can be restored after system crashes.

A_ Knowledge_Schema_Administrator

Responsibilities are:

- Design and install the B_Knowledge_Base_Schema.
- Update the B_Knowledge_Base_Schema whenever you feel an improved version could support the A_Knowledge_User better.

A_ Knowledge_Administrator

Responsibilities are:

- Review Practice Candidates.
- Decide whether to reject or accept a practice candidate.
- If a practice candidate is to be rejected, call S_Reject_Candidate.
- If a practice candidate is to be accepted, classify the items contained therein by adding associations of type R_Is_keyword_for and then call S_Accept_Candidate.
- If you feel that existing knowledge could be classified better, reclassify it by deleting or adding associations of type R_Is_keyword_for (export the package via S_Open_Practice_Candidate, edit the file **PackageRoot**/ **Structure**, ask S_Check_Package whether the new version is well formed, and – if so – save it via S_Save_Package with UPDATE = TRUE).
- If you feel that the B_Knowledge_Base_Schema currently in use could be enhanced, create a draft for a better schema and discuss it with the A_Knowledge_Schema_Administrator.

A_ Knowledge_User

Responsibilities are:

- Inform yourself about existing knowledge.
- Before starting to create a solution of some specific problem, query WissDB whether there are already solutions for similar problems.
- Export and reuse this knowledge where appropriate.

A_ Knowledge_Provider

Responsibilities are:

- Whenever in a project you see or create a solution you feel could be reusable, try to create a self-contained knowledge package.
- Edit this package until it is sufficiently self-contained to be accepted by S_Check_Package.
- Then call S_Save_Package to make this package a B_Practice_Candidate.

Data Views for C_WissDB_API

B_ Knowledge_Base

Definition owner is C_WissDB_DL_API

Value Specification:

- For each WissDB installation there is one and only one such value.
- The kind of knowledge it is to contain is discussed in D_(WissDB/ Design/ Data Model).
- It takes the form of a tree-like structured unit containing subunits that are instances of one of the following meta data entity types:
  - E_Process
  - E_Role
  - E_Practice
  - E_Result
  - E_KnowledgeItem
  - E_Aspect
- The treelike structure is a logical structure (not a physical one).
- In addition to the tree-like structure (given by attributes A_Loc) there is correlation structure implemented in the form of binary relations
  - R_Is_related_to
  - R_Is_keyword_for
- The set of valid values for the following domain types need to be configurable to some extent:
  - D_ItemType
  - D_PracticeType
  - D_ViewType
  - D_AbstractionType
  - D_UsageType
  - D_CorrelationType.

B_ Knowledge_Base_Schema

Definition owner is C_WissDB_API

Value Specification:

An object of type B_Knowledge_Base_Schema is created as follows:

- Let A_System_Administrator provide an empty B_Knowledge_Base. Then:
- Use S_Replace_or_create_DB_DomainValues to define valid values for
  - D_ItemType
  - D_PracticeType
  - D_ViewType
  - D_AbstractionType
  - D_UsageType
  - D_CorrelationType
- Convince yourself that now there is at least one valid value for each of these domain types.
- Use S_Update_Aspect_Knowledge to create a default classification schema (a set of aspect locators in use all having **1**/ **Aspect** as their shortest preLocator).
- Use S_Update_Process_Knowledge to create a suitable default process structure (a set of process and role locators all having **1**/ **Process** or **1**/ **Role** for their shortest preLocator).
- Convince youself that now there is at least on aspect, process and role locator in use that does not end with the name **Aspect**, **Process** or **Role** resp.

B_ Locator_Restrictions

Definition owner is C_WissDB_API

A value of type D_Locator ist a string **/N/** or **X/N/** or **A/** such that

- **N** is a name
- **X** is again a D_Locator
- **A** is a positive integer used to abbreviate a Project Locator.

A **Project Locator** is a D_Locator

- not starting with a number an not containing any of the names **Process**, **Role**, **Aspect**, **Result**,
- or a D_Locator of the form **X**/**Project**/**N** such that **N** is a Name and **X** is a D_Locator (**X** may start with a number).

Number 1 and 2 are reserved to abbreviate the Project Locators **/Default** and **/WissDB**.

Given a D_Locator **L** not starting with a number, let n(L,k) be the name on position k in this locator. Under the assumption that n(L,k) is one of the names **Project**, **Process**, **Role**, **Aspect**, **Result**, and assuming further that j > 0 is the smallest number such that n(L,k+j) is again one of these names, the following restrictions apply:

- n(L,k+1) must be different from n(L,k).
- n(L,k) and n(L,k+j) may both be **Result**. In this case however we must have j > 1.
- If n(L,k) = **Role**, then n(L,k+j) does not exist.
- If n(L,k) = **Aspect**, then n(L,k+j) does not exist.
- If n(L,k) = **Project** and n(L,k+j) = **Process**, then j = 1.
- If n(L,k) = **Process** and n(L,k+j) = **Role**, then j = 1.
- If n(L,k) = **Process** and n(L,k+j) = **Aspect**, then j = 1.

Given a locator L = **X**/**Process**/**N** or L = **X**/**Result**/**N** that is in use, the locator L/**Description** should also be in use (in other words: WissDB is to enforce that each Process or Result has a description).

Other entities may have a description but will not be forced to have one.

A general rule (WissDB will not be able to enforce) is: The locator of a knowledge item **X** should be prefix of the locator of another item **XX** if and only if

- **XX** is an essential part of **X**, or
- **XX** will make sense only in the context of **X**.

B_ Query_Specification

Definition owner is C_WissDB_API

Value Specification:

An object of type B_Query_Specification is

- the root **SR** of the folder that shall represent the knowledge package you expect to get as a search result
- and – directly under this root – an ASCII text file SR/**Selector** in which you specify the query to select knowledge from the WissDB B_Knowledge_Base.

The content of SR/Selector has to match the following pattern und is to be understood as a pair of filters such that the first one is specifiying items to be included into a **preliminary search result** from which then the **final search result** is derived by excluding items the user is not interested in:

```
   WissDB Knowledge Selector

   The first column of this file is - on a graphical user interface - presented in
   form of check boxes. A box is checked if and only if the corresponding line in this
   file here contains the minus sign in column 1.

   Lines not starting with a minus sign in column 1 are treated as being comment.


-  The Include Filter:
   Not checking the include filter is asking for all knowledge.

-  Type
   Not checking Type is the same as checking all these Type identifiers:

-  .  Process
   .  Description of Process
   .  Description of Role
-  .  Description of Result
-  .  Description of Practice Candidate
-  .  Solution Requirement
-  .  Solution Concept
-  .  Solution Code
-  .  Business
-  .  Technology
-  .  Advice
-  .  Information
-  .  Aspects

   Scope
   Not checking Scope is the same as checking all these Scope identifiers:

-  .  Result
-  .  Best Practice
-  .  Lesson Learned

   View
   Not checking View is the same as checking all these View identifiers:

-  .  Conceptual
-  .  Logical
-  .  Physical
-  .  Transferential
```

- Abstraction
  Not checking Abstraction is the same as checking all these Abstraction identifiers:

  . Solution
- . Template
- . Pattern
  . Strategy

- Aspects
  Not checking Aspects is the same as checking all these aspects (which will – in the WissDB Default Search Descriptor - be all aspects that are not part of any other aspect):

  . Bid Proposal Management
  . Project Management
- . Software Development
  . Software Support
- . Software Maintenance

  Note: As far as aspect locators L are shown here not starting with a number, they have to be understood as 1/Aspect/L (1/Aspect being the default classification schema).

  There may be project specific classification schemata as well.

- Projects
  Not checking Projects is the same as checking all projects (which will – in the WissDB Default Search Descriptor - be all projects that are not part of some larger project).

  . 1=Default
  . 2=Kogito

  Role Specification
  Not checking Role Specification is the same as listing all activities.
  {
  A Role is seen as a set of activities (= processes or subprocesses):

  The definitions here will restrict the search results to items that are part of Results produced by one of the following processes (resp. subprocesses thereof):

- 2/Process/Software Development/Test
- 2/Process/Software Maintenance/Test

  1/Software Development/Test
  1/Software Maintenance/Test
  }

  Note: Locators L shown her not starting with a number are to be understood as 1/L (for Results), 1/Aspect/L (for aspects), 1/Process/L (for processes).

- **The Exclude Filter:**
  {
  Not checking The Exclude Filter is asking the system to ignore this section.

  Excluding Aspects is reducing the set of keywords.

- Aspect/Bid Proposal Management/Tender Preparation

  Excluding a Process will exclude all Results produced via this process.

- Customer A/Process/Software Support
- Customer B/Process/Software Support

  Excluding a Result will exclude all parts of it:

- WissDB/Result/Test
  }

```
The Search Result: Included and excluded objects

This section is read only – it will be generated by S_Search_for_Knowledge.

Items checked via a minus sign in the first column of this section are items not excluded
from the search result. If too many items are shown to be excluded, the user can edit the
Exclude section and then again submit this selector to the S_Search_for_Knowledge service.

For excluded objects no structure is shown.
{
}
```

Rationale for this design:

Users need to get a good feeling what effect it will have

- when they augment the exclude section by adding specific locators
- or when they re-scope a tentative search result by
  - changing the role specification or
  - or by removing or restoring minus signs in column 1 of the search specification file.

Furthermore, given any knowledge package that came into life as a search result, users can always see what the query actually was.

B_ Default_Selector

This is the unique B_Query_Specification defined as follows:

- **The Include Filter** is checked.
- **The Exclude Filter** is checked (and does not contain examples).
- No other lines are checked.
- Section **Projects** is a List of all project locators currently in use.
- Section **Role Specification** is a list of all role and process locators currently in use.

B_ Search_Result

Definition owner is C_WissDB_API

Value Specification:

An object of type B_Search_Result is

- a B_Knowledge_Package **X**
- containing a B_Query_Specification in the file **X**/ **Selector**.

It may exist in form of a skeleton only (a **Skeleton** in this sense will only contain the **Structure** files and the **Selector** file).

As soon as the user feels satisfied with the scope of the search result (reflected by the skeleton of the package), he can expand it to the full package via S_Export_Knowledge.

B_ Knowledge_Package

Definition owner is C_WissDB_API

A knowledge package is the container needed to be transfer knowledge

- from the user to the knowledge base or
- from the knowledge base to the user's workstation.

The content of a knowledge package must be self-contained and self-describing.

Value Specification:

- See the section "Knowledge Packages" in D_(Design/ WissDB/ Data Model). Reading this document you will learn that the logical structure of a knowledge package is identical to its physical structure.

B_ Practice_Candidate

Definition owner is C_WissDB_API

Knowledge packages not yet merged into the knowledge base (but already stored there) are called **Practice Candidates**.

It is the duty of a A_Knowledge_Administrator to decide whether such a candidate is to be rejected or accepted. He may edit the package before accepting it.

Each B_Practice_Candidate is stored in the B_Knowledge_Base in form of a JAR file and a text file of any type (ASCII or MS Office), which have locators matching the pattern

Package/ D_Locator/ D_Name

Package/ D_Locator/ **Description/** D_Name

B_ Root

Definition owner is C_WissDB_API

Value Specification:

Each B_Root is a string that is either a B_URL in the sense of C_WissDB_DL_API or a path starting with a drive letter (meant to address some folder existing in the physical file system as it is seen by the WissDB user's workstation).

B_ Structure_Format

Definition owner is C_WissDB_API

Given the B_Root **X** of a B_Knowledge_Package, the file **X**/ **Structure** is an ASCII text file describing each item's structure in the following format (each minus sign in column 1 following an empty line, each dot in column 4 followed by two spaces):

```
-  ItemLocator
   .  AttrName: Value
   .  AttrName: Value
```

Here an **AttrName** is the name **a_N** of a (concrete or logical) attribute shown in the WissDB ERD found in section 2 of D_(WissDB/ Design/ Data Model).

If **X**/ **Structure** is a file created by S_Check_Package, there will be no **AttrName** lines, but the file will contain a header of the form

```
   Package Structure (as of YYYY.MM.DD.HH.MM):
```

Before submitting such a package to S_Check_Package or S_Save_Package, the user himself will have to add **AttrName** lines representing relationships of type R_Is_related_to. For each of them the **AttrName** is to be a presentation of a currently valid D_CorrelationType value.

As far as relationships of type R_Is_keyword_for are to exist, they need to be added in form of sections taking the form

```
-  AspectLocator
   .  ItemLocator
   .  ItemLocator
```

If **X**/ **Structure** is a file created by S_Export_Descriptions, **AttrName** lines are shown, and the file will contain a header of the form

```
   Structure of Search Result (as of YYYY.MM.DD.HH.MM):
```

and a last section

```
   Item Classification:
```

containing all R_Is_keyword_for relationships currently stored in the B_Knowledge_Base. The last line of this section will always be

```
   end Item Classification in Search Result as of YYYY.MM.DD.HH.MM
```

Services offered by C_WissDB_API

This section contains the logical design of the WissDB API API. We start by describing important requirements on the form this API has to take:

R_ How to describe and report Application Errors

Each service is specified with two specific parameters AppErrors and ReturnCode:

- **Out: AppErrors**
- **Out: ReturnCode** is one of the following values:
  - RC_ok
  - RC_syserror
  - RC_seeAppErrors

Depending on the programming langue used, these two parameters may be implemented quite differently. In Java, e.g. at least RC_syserror would best be implemented in form of a runtime exception, whereas RC_seeAppErrors might be implemented differently from case to case:

- If an exception such as ObjectNotFound is enough error description, to throw an exception implementing both the returncode RC_seeAppErrors and also the AppErrors would be fine (and should be preferred).
- If however syntax errors (or logical errors hard to explain) are detected somewhere, the AppErrors need to be a dynamically generated text the application could then write to either a file or the console.

A value of type **AppErrors** is a string containing one or more text sections of the form

AppError in S_x: ErrorType: Diagnostic data

or

Warning out of **S_x:** any text

such that **S_x** is the service activated by the user or an application of the API API (it must not be the name of an auxiliary method that is part of the hidden implementation of this service: users and applications are to see the system they use as a black box).

The service is to return RC_seeAppErrors if and only if parameter AppErrors is describing at least one AppError.

The service should describe as many AppErrors as possible before returning.

S_ Check_Package

Component:	**S_Check_Package** part of C_WissDB_API
Abstract:	This is the function allowing the user to ask the system in what respect a given knowledge package is not yet well formed. What it means for a package to be **well formed** is specified in section 2.2. of D_(WissDB/ Design/ Data Model) und the conceptual point if view. From a more formal point of view there is the additional condition that the content of the file **PackageRoot**/ **Structure** is to be text in B_Structure_Format such that all items and aspects named therein actually exist. Aspects are searched for first under the **PackageRoot** and then – if not found there – in the B_Knowledge_Base. Parameters: – **In: PackageRoot** The B_Root of the knowledge packages the user wants the system to store. – **Out: AppErrors** If the package was not well formed, users should find a description here telling them how to make the package well formed. – **Out: ReturnCode** is one of the following values: – RC_ok – RC_syserror – RC_seeAppErrors
Has to:	Check whether the given tree of files can be seen as a well formed knowledge package. If it is not well formed, let AppErrors tell the user what they are to change in order to make the package well formed.
Because of:	End users need a way to check how far a package is already well formed.

S_ Save_Package

Component:	**S_Save_Package** part of C_WissDB_API
Abstract:	This is the function allowing the user to store or update in WissDB a Practice Candidate (a B_Knowledge_Package not yet accepted as a result or practice instance worth to keep). Parameters: – **In: PackageRoot** The B_Root of the knowledge package the user wants the system to store. – **In: Loc** This is a Project Locator suggested by the user. The candidate's A_Loc value is to become **package**/ **Loc**/ **pack.jar** (WissDB is to create this JAR archive). – **In: Update** This is a Boolean value. FALSE means: This is a new package that must not replace existing data. – **In: Submitter** This is either NULL or a string representing an e-Mail address. – **Out: AppErrors** If the package was not well formed (and was therefore not accepted), users should find a description here telling them how to make the package well formed. Describe an AppError here if **Submitter** = NULL and **Update** = FALSE. Describe an AppError if the given **PackageName** does not have the form of a Project Locator. – **Out: ReturnCode** is one of the following values: – RC_ok – RC_syserror – RC_seeAppErrors
Has to:	Save the package, or the new version thereof, if and only if S_Check_Package says that it is well formed.

	In case of **Update** = TRUE, saving the command may overwrite a previous version (if there is one). Practice candidates stored but not yet accepted or rejected can be found via the service S_Search_for_Candidates.
Because of:	End users need a way to hand over practice candidates to the WissDB system. A_KnowledgeAdministrator may want to edit a package one or more times before it can be merged into the B_Knowledge_Base.

S_ Open_Practice_Candidate

Component:	**S_Open_Practice_Candidate** part of C_WissDB_API
Abstract:	This is the function allowing the user to copy to his workstation a currently existing practice candidate. Parameters: – **In: PackageName** The name of a previously stored practice candidate (S_Search_for_Candidates can tell you which candidates currently exist). – **In: PackageTo** The absolute path to the place where the user wants to work with the current version of practice candidate (a knowledge package). – **Out: Submitter** Is a string representing an e-Mail address: a value E_Candidate.A_from. – **Out: AppErrors** Is to describe an AppError if the package could not be found or could not be written to the place given in **PackageTo**. – **Out: ReturnCode** is one of the following values: – RC_ok – RC_syserror – RC_seeAppErrors
Has to:	Find and extract the package identified by **PackageName**.
Because of:	End users need a way to hand over practice candidates to the WissDB system. A_KnowledgeAdministrator may want to edit a package and the store a new version not yet good enough to be merged into the B_Knowledge_Base.

S_ Reject_Candidate

Component:	**S_Reject_Candidate** Part of C_WissDB_API
Abstract:	This is the function asking WissDB to reject a previously stored (but not yet accepted) practice candidate. Parameters: – **In: PackageName** The name of a previously stored practice candidate. – **Out: AppErrors** Is to contain one or more AppError description if the package could not be found or if the mail to send could not be sent. – **Out: ReturnCode** is one of the following values: – RC_ok – RC_syserror – RC_seeAppErrors
Has to:	Find and forget the package identified by **PackageName**. Send an e-mail to the person who made this package a knowledge candidate (the address being stored as a value E_Candidate.A_from).
Because of:	A_KnowledgeAdministrator may want to reject a package (because the knowledge it represents is not worth keeping, or because the results found therein are not documented good enough).

S_ Accept_Candidate

Component:	**S_Accept_Candidate** part of C_WissDB_API
Abstract:	This is the function asking WissDB to merge the content of a practice candidate into the B_Knowledge_Base. Parameters: – **In: PackageName** The name of a previously stored knowledge package. – **Out: AppErrors** Is to describe AppErrors if the package could not be found or could not be merged into B_Knowledge_Base. Is to contain a warning if the mail to send could not be sent. – **Out: ReturnCode** is one of the following values: – RC_ok – RC_syserror – RC_seeAppErrors
Has to:	Find the practice candidate identified by **PackageName** and merge its content into the B_Knowledge_Base. Then forget the candidate. Send an e-mail to the person who made this package a practice candidate (the address being stored as a value E_Candidate.A_from).
Because of:	As soon as A_KnowledgeAdministrator is satisfied with the content and the structure of a knowledge package that should be kept in form of practice instances, process schema, or indexing schema he needs a function to merge this content into the knowledge base (so that, from now on, it will be retrievable via the service S_Search_for_Knowledge).

S_ Search_for_Candidates

Component:	**S_Search_for_Candidates** part of C_WissDB_API
Abstract:	Using this service the user can ask WissDB to identify existing knowledge packages. These are, by definition, packages stored but not jet accepted or rejected. Parameters: – **In: Selector** A possibly empty string. It is to say: “Get me a list of all candidate **X** found in B_Knowledge_Base.E_Candidate such that **Selector** is a substring of either **X**.a_Submitter or **X**.a_Loc”. – **Out: Locators** A set of objects of type E_Candidate.(a_Loc, a_since, a_from). – **Out: AppErrors** – **Out: ReturnCode** is one of the following values: – RC_ok – RC_syserror – RC_seeAppErrors
Has to:	Search for packages stored but not yet rejected or accepted.
Because of:	WissDB users need a service to search for existing packages (and see their status).

S_ Search_for_Knowledge

Component:	**S_Search_for_Knowledge** part of C_WissDB_API
Abstract:	This is the service supporting knowledge retrieval. Parameters: – **In: PackageRoot** The absolute path to the place where users want to have a package representing the knowledge to be retrieved. – **In: QuerySpec** A value of type B_Query_Specification existing at **PackageRoot**/ **Selector** in form of an ASCII text file. – **Out: PackageStructure** The search result, if seen as a knowledge package, has structure. This structure shall be delivered as the new content of the file **PackageRoot**/ **SearchResult**. – **Out: AppErrors** Is to describe one or more AppErrors if – the PackageRoot folder does not exist, – the query specification, if missing, could not be created, or – the query specification, if found, was not well formed. – **Out: ReturnCode** is one of the following values: – RC_ok – RC_syserror – RC_seeAppErrors
Has to:	Search the B_Knowledge_Base using the B_Query_Specification ▪ either found as a file **PackageRoot**/ **Selector** ▪ or created there as a copy of the B_Default_Selector (if at least the folder PackageRoot exists).
Because of:	WissDB users need a service to search for – and to retrieve – practice knowledge. Users not knowing how to create a B_Query_Specification should be shown a well explained example (start with the B_Default_Selector to create such an example).

S_ Export_Descriptions

Component:	**S_Export_Descriptions** part of C_WissDB_API
Abstract:	Using this service the user can ask WissDB to export the Description items of the knowledge found via S_Search_for_Knowledge. Parameters: – **In: PackageRoot** A B_Root **X** such that the file **X**/ **Selector** exists and contains a not empty last section **Included and Excluded Items**. – **Out: Exports** Is a lexicographically sorted sequence of strings representing the locators for the descriptions found and exported. – **Out: Structure** Is an updated version of a last section “Package Structure” added by WissDB in the **X**/ **Selector**. This section is to describe in B_Structure_Format all the structure of the knowledge items listed as **Included** in **X**/ **Selector** (attributes a_x in the sense of the Entity Relationship Diagram found in the section 2 of D_(WissDB/ Design/ Data Model). – **Out: AppErrors** Is do describe an AppError if the **PackageRoot** could not be found or if at least one description file could not be created (or updated) in this tree. – **Out: ReturnCode** is one of the following values: – RC_ok – RC_syserror – RC_seeAppErrors
Has to:	For each locator **L** checked in the section **Included and Excluded Items** of **X**/ **Selector** and ending with /**Description**, create – if content is found in this node – the file **R**/**L**/Description.**ext**, where **R** is the **PackageRoot** and **ext** is the file’s type.
Because of:	WissDB users need a service to export knowledge items found via one or more calls of S_Search_for_Knowledge.

S_ Export_Knowledge

Component:	**S_Export_Knowledge** part of C_WissDB_API
Abstract:	Using this service the user can ask WissDB to fully export the knowledge found via S_Search_for_Knowledge. Parameters: – **In: PackageRoot** An absolute path **X** into the file system such that the file **X**/ **Selector** exists and contains a not empty last section **Included and Excluded Items**. – **Out: AppErrors** Must not be empty if the result file could not be found or if the package described therein could not be produced completely. – **Out: ReturnCode** is one of the following values: – RC_ok – RC_syserror – RC_seeAppErrors
Has to:	Copy to **X** all content associated to locator values found checked in the section **Included and Excluded Items** of **X**/ **Selector**.
Because of:	WissDB users need a service to export knowledge items found via one or more calls of S_Search_for_Knowledge.

S_ Describe_Knowledge_DomainValues

This is currently only a wrapper around C_WissDB_DL_API.S_Describe_DB_DomainValues.

S_ Replace_or_create_Knowledge_DomainValues

This is currently only a wrapper around
C_WissDB_DL_API.S_Replace_or_create_DB_DomainValues.

S_ Rename_Project

This is currently only a wrapper around C_WissDB_DL_API.S_Set_Alias.

S_ Testsupport_New_Knowledge_Base

This is currently only a wrapper around C_WissDB_DL_API.S_Testsupport_New_DB.