– Diversity Workbench –
A biodiversity component framework
Draft Version 0.3

by G. Hagedorn
10. December 2002

Preface

As the version number '0.3' indicates, this publication is a preliminary draft containing fragments for discussion. It has been made publicly available to facilitate the discussion about an internationally accepted, public component framework definition for biodiversity information. I appreciate any feedback and criticism!

Table of contents

Introduction 1
Current practices 2
Levels of modularization 4
Proposed modular interface components 6
Design Principles 8
Interface implementation issues 9
Interfaces, code classes, and content instances 10
Versioning components of the framework 12

Introduction

The Diversity Workbench is conceived as a set of applications and components for building and managing biodiversity information, each of which concentrates on a particular domain of biodiversity information and each of which can provide services as a component to the other components.

The focus of development in biodiversity informatics is generally on applications for information building. This is a peculiar contrast to many other fields of applied informatics, where information retrieval and application are at the center of research and where the basic information is readily available. Perhaps 60-80 percent of biodiversity information is not only not digitized, but not available at all (see Fig. 1), and the efficiency of biodiversity research must be increased to fill the knowledge gap.

The modularization of applications dealing with biodiversity information into exchangeable components is a demanding task. Given the complexity of biodiversity information, it is, however, an essential prerequisite to allow the successful implementation of adequate information systems with the limited resources that are available.

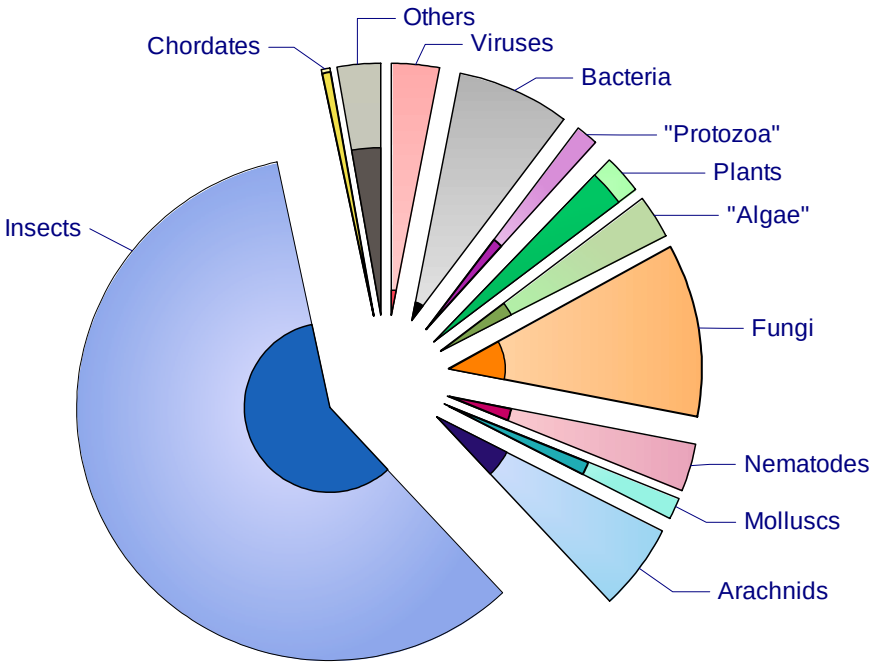

Figure 1. Species diversity of major groups. The area of the segments indicates the estimated total number of species living on earth. The area of the darker colored inner segments indicates the number of known species. For example, only ca. 5% of fungi, 10% of algae, 11.5% of insects are already known, but ca. 96% of chordates and 84% of (higher) plants.

The following attempts to define what is understood under a ***component framework***.

> ***Definition:*** *A* ***component framework*** *is a concept used to subdivide a complex information landscape into modular information models that interact through defined interfaces. The Diversity Workbench is an attempt to implement such a framework for the field of biodiversity information.*

This document attempts to inform about some of the requirements identified, and the strategies chosen. It is currently by no means comprehensive, but provides fragments for a future, more comprehensive documentation.

Current practices

Substantial efforts are made in various groups to build biodiversity information applications. The following diagram (Fig. 2) displays a personal estimate of the author about the development focus of a selected applications. The estimate should by no means be seen as an objective comparison. Only in the cases of DiversityWorkbench, BioLink, Specify PANDORA/Pankey, and Nomencurator could the information model be studied, the remaining models were not available to the author. Neither should the exclusion of some develop-

ments be considered a statement that it is less important; it may simply be less well known to the author. And finally, in the case of the Workbench components the quality of the information model is not related to actual release stability of applications; for example, for Diversity-Taxonomy (names, synonymy, and taxonomy) no application exists at all.

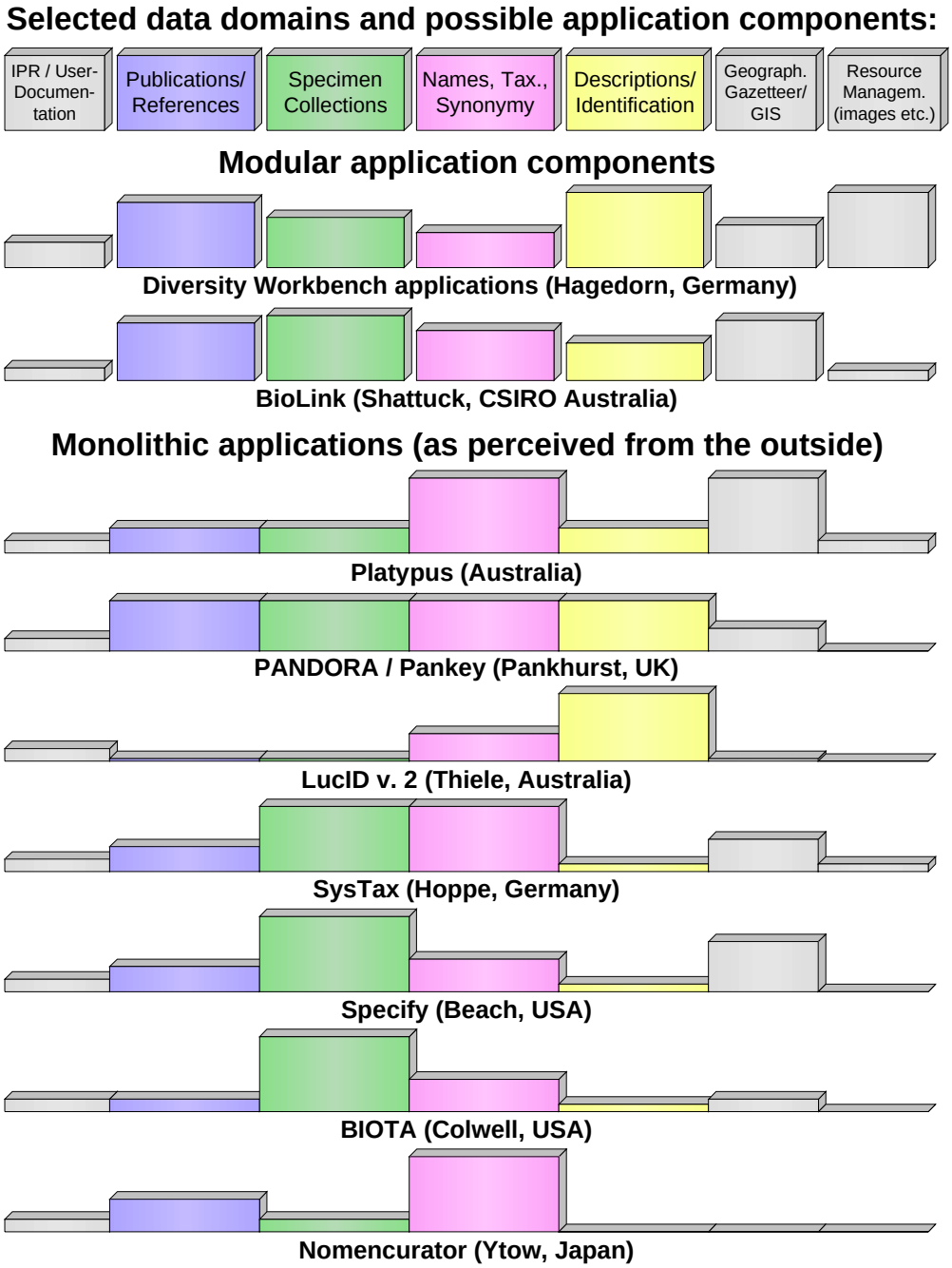

Fig. 2: Selected biodiversity informatics information building applications and with an estimate about their development focus. (Abbreviations: IPR = Intellectual Property Rights, GIS = Geographical Information Systems.)

The intention of the comparison is to show that different development teams focus on different aspects of biodiversity informatics. In addition, each team also has the burden to implement minimal-requirement versions of most other aspects. The benefit of an accepted public

component framework would be that each development team continues to concentrate on its primary topic, but then can use each others efforts to provide the remaining infrastructure. If a common framework and interface definitions exist, the components would be interoperable and exchangeable. The concentration on a single domain would lead to higher quality products.

Furthermore, data would no longer have to be duplicated in several applications if multiple applications are required within an institution. This is currently the case if one application is used to manage the specimen collection and another for taxonomic research, but both carry an internal minimal literature reference component.

Currently, most applications are constructed monolithic and completely are non-interoperable. The term monolithic is used here in the sense that they completely hide the internal complexity (which may or may not be modularly designed in the source code) and present themselves to the outside with a single interface.

Some application suites have a visibly modular component design, notably DiversityWorkbench and BioLink. However, they do not use a common framework or interface definitions and consequently are not interoperable among each other.

Levels of modularization

Modularization of biodiversity information software can be achieved on three different levels:

Modularization of data storage

The complex field of biodiversity informatics includes information about the naming of organisms, the descriptions of organisms (properties useful for identification as well as other properties), the geographic distribution of organisms and interaction of organisms. Further, bibliographic, historic, geographic, and resource management data (including natural history collections as well as digital information collections) must be managed. Rather than creating highly complex, integrated information models (monolithic models), the information models should be created separately for each domain of biodiversity information and the interactions between these models should be defined.

> **Current development:** The analysis how to subdivide the complex information into modular information models has begun in the context of the BIOLOG funded German GLOPP project. This needs to be continued. The resulting models should be readily understandable and should minimize the interaction between components.

Modularization of functionality and code

The programming code that acts upon each domain of biodiversity information for which a modular information model should also be created as a component. Normally the code itself will have a internal modular structure. The "component module" will correspond rather to a JAVA package or a .NET assembly, consisting of multiple internal modules.

To allow the interaction between components, an object model should be created to access, manipulate, and analyze the biodiversity data. The object model can provide a uniform programming interface to obtain information from linked information models. To reduce the design complexity and allow as many implementation choices as possible, the component framework model should only define interface classes for components.

Using an abstracted simplified object oriented programming interface will generally be preferable to accessing the data of the various information models directly (e.g. through ODBC or JDBC interfaces). However, for certain analyses a direct linking of the data using database functionality rather than object oriented programming models will be desirable. Pure object oriented models, at least those without multiple inheritance, enforce a hierarchical view on the data, that is not always appropriate to the multidimensional reality of biodiversity data. A relational database can usually handle multidimensional data considerably better than current object oriented models.

Example: The Taxonomy application could call a reporting method in the Reference management application to provide an xhtml formatted reference citation in a specified format for a reference ID that has been stored together with taxonomic information.

> **Current development:** The modularization and generation of simple object oriented interfaces has started in the GLOPP project in an *ad-hoc mode*. It needs to be carefully analyzed and redesigned to become a stable model. The design of comprehensive object oriented models for the entire field of biodiversity information has also started in an other project (see e. g. the models by Guillaume Rousse, http://lis.snv.jussieu.fr/~rousse/recherche/index.html).

Modularization of components or applications

Using a detailed object model allows often very intricate interactions between components. However, it also ties components very tightly together and requires detailed knowledge about each component.

The problem with this is technology change and the evolution of information models. It is my believe that information models (relational models as well as object models) will need continuous refinement for at least another decade. Enforcing a detailed compatibility model for all attempts to implement a given domain of biodiversity information in a way that it can cooperate with other applications and components is therefore considered undesirable. It is

unlikely that two implementations of, for example, the bibliographic reference management component will be able to present the same detailed object oriented interface.

Componentization is a proven technique for improving application manageability, by isolating technology change. Components allow incremental rather than all-at-once redesign or biodiversity information application that cooperate in a component framework (e. g., the DiversityWorkbench).

Every effort should be made to minimize the interface definition that is required for components to interact in the framework. For some components this can be achieved through an object oriented interface that does not include a user interface component. An example might be User documentation or a geographical gazetteer. Many components will, however, have complex data structures that need to be accessed by the user through a query interface. Such an interface usually requires detailed knowledge about the underlying data structures.

Example: The Taxonomy application could directly call the Reference management application to ask the user to select the appropriate reference. The potentially complex query, as well as the potential need to enter a new reference if the reference in question is not yet in the database, can be handled internally by the Reference management application. The interface needs only concern itself with the returned ID number of the selected reference, or with cancellation of the dialog.

> **Current development:** The Diversity Workbench uses the approach of using user interface components in the current implementation. However, the current implementation works only because all components of the Diversity Workbench are based on the same development tool. Much work is necessary to generalize this approach, to define the interface in a rigorous way rather than ad-hoc. It is envisaged that the interaction could be implemented through the use of xml-SOAP remote procedure call mechanisms, and that it is potentially possible to define interactions between user interfaces that employ html forms.

Attempt of definition for component:

> ***Definition:*** *A component is defined here as a combination of data storage and application code, that can largely be used independently of other components (although it may require the service of certain other components).*

Proposed modular interface components

Figure 2 above already uses a selection of data domains to define and delimit interface components. The following diagram (Fig. 3) shows further examples of reusable and interoperable interface components for biodiversity information retrieval, building and appli-

cation. Note that a given application will often provide many interfaces (e. g. all of specimen collection management and look-up thesauri).

Also, applications do not need to follow the interface model internally. As long as it provides all the interfaces defined in the Workbench framework for the components it encompasses, the internal implementation is irrelevant, and it can function as multiple components in regard to other applications. For example, an application may integrate Nomenclature, Synonymy, and Taxonomic hierarchy. The definition of small interface classes is intended to allow flexibility in the choice of implementations for both service providers and service consumers (see "Interface implementation issues", p. 9).

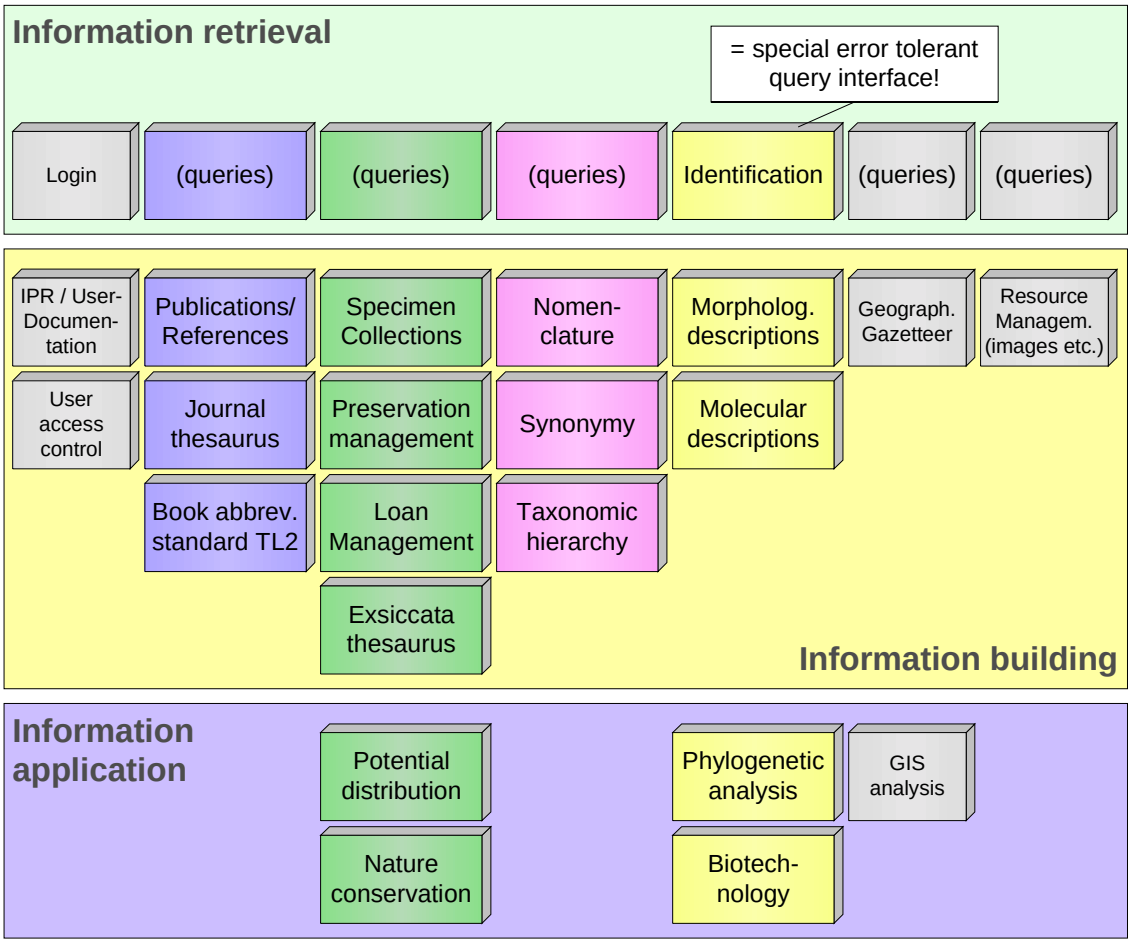

Fig. 3: Examples of components for biodiversity information retrieval, building and application.

The components shown in Fig. 3 are only examples. The definition of a complete component framework is so far neither completely analyzed, nor are formal interfaces defined in cases where the analysis already occurred for the purpose of developing the DiversityWorkbench components. Until substantial discussion between different groups has occurred and separate implementations have tested issues of interoperability, the framework will constantly evolve. The current development of the DiversityWorkbench by G. Hagedorn and coworkers is shown

in Fig. 4. The various components are in different stages of development and serve as a testing environment rather than as stable application components.

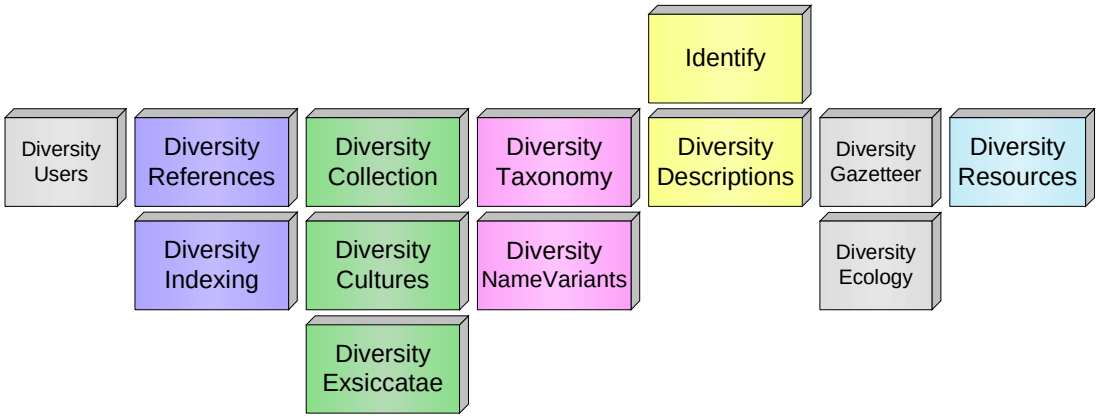

Fig. 4: Some DiversityWorkbench components currently under development.

Design Principles

The following principles outline some of the design principles that apply to all information models for components of the DiversityWorkbench component framework. Each component should:

- define one or several primary object types or classes
- provide a public, machine-readable, permanent object identifier (primary key) to these objects
- provide a unique human-readable object description (which may change as the underlying object data are modified and can therefore not be used as the primary object identifier)
- provide a minimal interface definition for public objects ('black box')
- document an internal object or entity-relationship model

Components using the internal model rather than the interface would be not exchangeable. However, properly implemented, the base functionality would be framework compatible and extended functionality would only work with specific component implementations.

Further, to guarantee the scientific quality of data, each component should:

- cite each piece of information from a published source (including natural history collection objects) or attribute direct statements to a person responsible for data entry (in effect making the database the primary place of information).
- contain published information as close as possible to the original source (obvious general spelling errors may be corrected, but no changes should be introduced requiring interpretation or specific background knowledge)

- separate original information from interpretation (e. g. changing misapplied or synonym names to accepted names)
- allow an unlimited number of (possibly contradicting) interpretations to support the ongoing refinement of information interpretation by several generations of researchers

---

Interface implementation issues

The component framework model is a model of interface classes. It can be used as a reference model to make decisions about the modularization of domain logic and information models for persistent storage, but it does not enforce it. For example, the internal implementation of the Nomenclature, Synonymy, and Taxonomic Hierarchy interface classes may either be modular or monolithic. As long as the interface components are implemented according to the framework model, this does not make a difference to another component consuming a synonymization service from Synonymy (see Fig. 5).

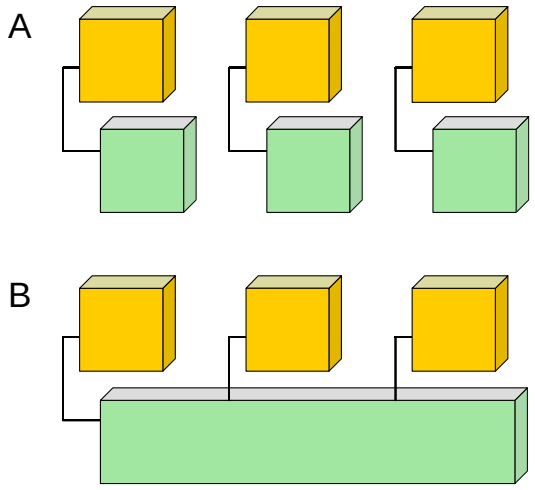

Fig. 5: A) The internal implementation follows the interface component model. B) The internal implementation is monolithic, but implements separate interface classes. For an external consumer of component services both implementation variants function identically. Interface classes are orange, implementation classes green.

If the internal modularization does follow the component framework model and several components are developed together, the developer still has several implementation choices. It will generally be advisable to use those methods defined in the framework also internally, to avoid duplicate interface implementations and testing. However, if for example, the normal communication between components is based on xml SOAP, the same methods can also be used by communication methods that are native to the development tools used (Fig. 6).

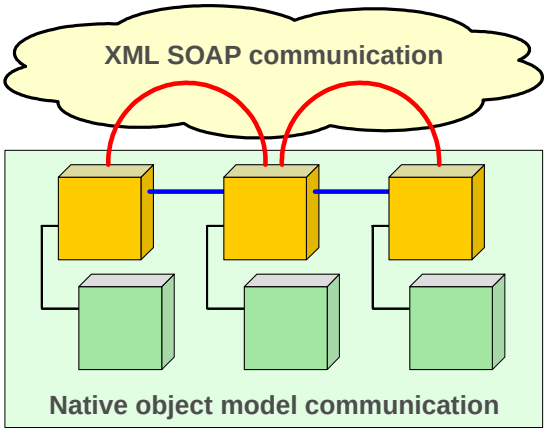

Fig. 6: The implementation may use the component model for part of the internal communications, but may use a different communication layer. In the example, the external communication occurs through xml-SOAP, but the internal communication through a native method (blue connections). Interface classes are orange, implementation classes green; compare also Fig. 5.

Furthermore, it is possible to implement additional interfaces providing methods that are not exposed through the framework interface model. As discussed above, the framework attempts to treat components as "black boxes", hiding internal complexity and allowing for a variety of different solutions to a problem. However, communication within components developed by the same team may wish to exploit detailed knowledge of the internal object model of the components.

Care must be taken in this case, that the additional methods are either optional, or that it is well documented that certain components are exchangeable only as a package. For example, consider again the application implementing several taxonomical components (Nomenclature, Synonymy, and Taxonomic Hierarchy), requiring services from a literature reference component. The reference component may only need part of this functionality and any combination between implementations of these components may be possible. However, it may not be possible to exchange the nomenclature component of the combined taxonomic application with a better nomenclature component.

A good design would derive the rich interface classes from the minimal interface classes defined in the component framework. Consumers of the interface could then provide polymorphic variable, testing whether the basic or the extended interface is present.

Interfaces, code classes, and content instances

The Workbench model assumes that compatible components can be freely combined to provide a working environment. These components must identify themselves in multiple ways. They must identify:

- the component framework (e. g. "DiversityWorkbench"),
- the component interface class (e. g. component for "Reference management", "Resource management", or "Collection management"),
- the implementation (application or component developers signature)
- the content instance (implementations of a component may be used by multiple projects and filled with different content)

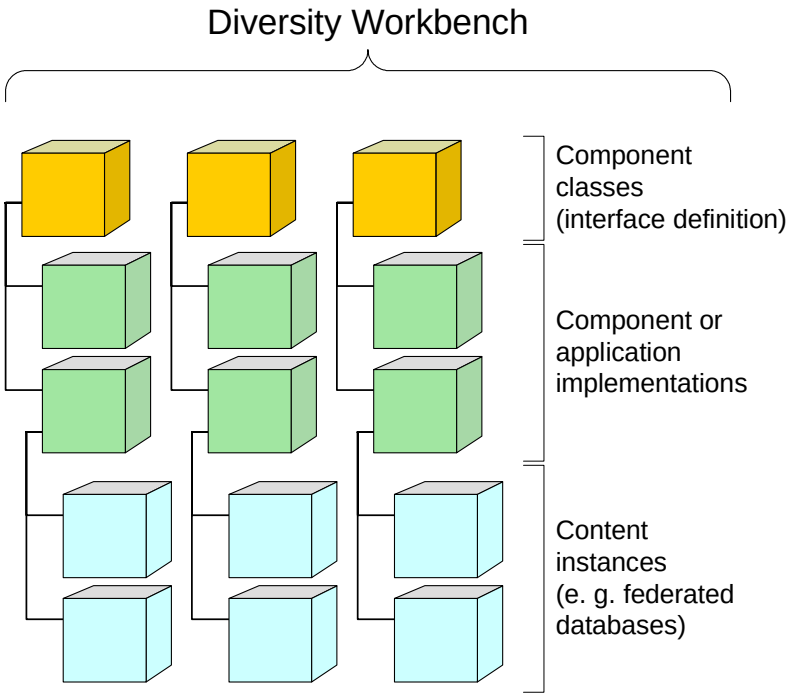

Fig. 7: Multiple implementations and multiple content instances for each component are possible within a component framework.

Figure 7 illustrates these levels. The framework and the possible interfaces classes would be fixed by the definition of the component framework itself. They are only tested to verify the compatibility (esp. if different versions of the framework exist). However, multiple implementations and content instances can coexist within the framework. For these both a signature and versioning support must be provided.

A signature defines the identity of a component. The implementation signature and version are irrelevant if only the base functionality of the component framework is used. In this case only the framework identity and version needs to be tested. However, if implementation-specific extended functionality is used by a consuming component (see "Interface implementation issues", p. 9 above), both must be tested.

In contrast, the content signature is always relevant to verify the identity of the an object identifier that needs to be resolved. The same object identifier may exist in multiple content instances and may refer to different objects. Object identifiers that are globally unique among all content instances would require communication between multiple content instances and a

central registry to do so. This is considered undesirable. However, the combination of the signature value of a content instance and an object ID is assumed to provide a globally unique object identifier. To achieve this a global registry of content instance signatures is desirable in the longer run, but choosing random signature IDs will also provide an acceptable level of uniqueness.

Example: To refer, for example, from a taxonomic component to a reference defined in a literature or citation reference component, the following data items about the reference object are stored in the taxonomic component:

- a technical object identifier representing the reference
- a human-readable description of the reference that may be cached by the taxonomic component to provide a fast overview report, esp. for screen display
- the signature of the content instance from with the object identifier is derived.

Note that the reference detail (page number, figure or table number) is not linked to the reference component, but directly stored within the taxonomic component. A validation method for common mistakes (at least in the case of simple page numbers) can easily be implemented if this is desired.

Versioning components of the framework

Most components will have a persistent storage component that is managed separately from the application logic in a database management system. Although this separation has many advantages, it also complicates the versioning. Data components can have three different version numbers:

- The version for the application logic of the component
- The version of the schema or information model used to store the data
- The version of the content in the database

In practice, the second level will normally not be of interest to the consumer. The compatibility of the application code and the information model will automatically be tested during the initialization of the component interface, using signature and version information for both parts. Note that the signature of the code and the information model is required to be identical in the current DiversityWorkbench implementations of the Workbench model.

However, components that want to utilize another component need to be aware of the interface definition compatibility and of the content signature and version.

Forward and backwards version compatibility

If components shall interact freely, it is necessary that the interface definitions are forward and backwards compatible. The following example illustrates the problems.

Both the collection and the references component consume services from the geographical gazetteer component. The initial situation may be that all 3 components are in version 1:

Diversity References	**Diversity Gazetteer**	**Diversity Collections**
Version 1.0	Version 1.0	Version 1.0

A few months later, the collections component is upgraded to a new version, which requires the version 2 of the gazetteer as well. A new version of the references component is, however, neither available nor required based on the reference component itself:

Diversity References	**Diversity Gazetteer**	**Diversity Collections**
Version 1.0	Version 2.0	Version 2.0

Ideally, the references component should now still be able to interact with the interface of the Gazetteer.

Proposed solution providing forward and backwards version compatibility

The solution that was chosen to provide for this functionality is based on two concepts:

Firstly, each component provides a primary interface class that may be extended through the use of additional properties or methods, but which is never changed. Also, the base interface class will never change its name (to avoid namespace conflicts, the name is based directly on the component name, with "interface" appended, e. g. "GazetteerInterface"). This primary component interface mainly provides basic information about a component (component name, version, identity signatures, copyright information, initialization success and errors, etc.). Most of these methods are generic to all components and derived from an abstract class in the framework.

Secondly, any complex methods which are more likely to change are provided through version specific derived interface classes. The version number is part of the class name. For example, the Gazetteer may provide information about a geographical name through a class Gazetteer10_GeoName. The version-specific classes can be accessed and instantiated through the use of methods provided in the base interface class.

If a new version of the Gazetteer provides additional functionality, the developer has two options:

a) to extend the Gazetteer10_GeoName through the use of additional properties or methods without changing existing properties or method
b) to introduce a new class Gazetteer20_GeoName, while preserving the Gazetteer10_GeoName class for the purpose of backward compatibility.

In the first case, the programming of Diversity Collection does not need to be upgraded, but the application must test the version number of the Diversity Gazetteer to make sure that the additional methods are available. The Interface provides a standard method InitializeOnlyWithMinimumVersion to prevent initialization for any interface that has a smaller version number.

In the second case, the new interface can be redesigned and old errors or shortcomings can be prevented. However, all components that use this interface must be adapted to use the new functionality.

Note that the version number in the interface classes is thus not necessarily identical with the version number of the component. It should rather be read as "Interface introduced starting with version X.x".

Information model for persistent storage of interface version and compatibility information

The following information model describes a proposal for two entities to hold versioning and compatibility information:

(Note: the entities shown below are used by existing DiversityWorkbench components. They use the term "module" as a synonym of "component", something that should be changed when designing a public standard.)

WorkbenchInterfaceDescription

This entity contains two records, describing the title, version number, and description of schema (= information model) and content (= data collection) of a database component (Workbench Framework 1.0).

Attributes and indices of the entity 'WorkbenchInterfaceDescription'

Name	Type	Description / Default value & validation	Rqrd.	Index
Type	Text	Only two records possible: 'Content' (defined by the user) and 'Schema' (= information model, defined by developer, must be compatible with the application). *Default value: "Content"; Validation rule: "Schema" Or "Content"* *Values currently restricted to:* Content	R	I (P)
MajorVersion	Long	The major version number of the schema or content data collection. Example: '2' for version 2.1. *Default value: 1*	R	-
MinorVersion	Long	The minor version number of the schema or content data collection. Example: '1' for version 2.1. *Default value: 0*	R	-
Revision	Long	The revision version number of the schema or content data collection. Example: '5' for version 2.1.0005. Displayed only if > 0. *Default value: 0*	R	-
VersionSuffix	Text	An optional string to identify special versions. Examples: 'beta', 'rc' = release candidate, etc. *Validation rule: Is Null Or Not (Like ' *' Or Like '* ' Or Like '* *' Or Like '*' & Chr(10) & '*' Or Like '*' & Chr(13) & '*'), validation message: Do not start or end with a blank and do not include double blanks or new line characters!* *Examples values (any other values may be added):* beta = testing version; rc = release candidate; rc-1; rc-2 = second rc; alpha = unfinished component; sr-1 service release 1; sr-2 service release 2	-	-
Signature	Long	A number that uniquely identifies a schema or content data collection (used by other components to identify object identifiers as belonging to this data collection; content signature values must be 1-524287, schema 1-4095). *Default value: CLng(Rnd()*524287), validation message: The content signature must be between 1 And 524287.*	R	-

Attributes and indices of the entity 'WorkbenchInterfaceDescription'				
Name	**Type**	**Description / Default value & validation**	**Rqrd.**	**Index**
Title	Text	A title for the schema or content data collection. Example: 'Mycological Literature collected by the Mycology.Net initiative'. *Validation rule: Not (Like ' *' Or Like '* ' Or Like '* *' Or Like '*' & Chr(10) & '*' Or Like '*' & Chr(13) & '*'), validation message: Do not start or end with a blank and do not include double blanks or new line characters!*	R	-
Description	Memo	An optional description of the entire information schema or data collection (will be displayed in the 'About' dialog box). *Validation rule: Is Null Or Not (Like ' *' Or Like '* ' Or Like '* *' Or Like '*' & Chr(10) & '*' Or Like '*' & Chr(13) & '*'), validation message: Do not start or end with a blank and do not include double blanks or new line characters!*	-	-
CopyrightStatement	Text	A copyright statement concerning the data. *Validation rule: Is Null Or Not (Like ' *' Or Like '* ' Or Like '* *' Or Like '*' & Chr(10) & '*' Or Like '*' & Chr(13) & '*'), validation message: Do not start or end with a blank and do not include double blanks or new line characters!*	-	-
PublicLicense	Text	Type of public licence (none, GPL, LGPL, etc.). *Default value: "GPL"*	-	-
Values currently restricted to:	None	No public license		
	GPL	GNU General Public License Vers. 2		
	LGPL	Lesser General Public License for code libraries		
Authors	Text	A list of author(s). *Validation rule: Is Null Or Not (Like ' *' Or Like '* ' Or Like '* *' Or Like '*' & Chr(10) & '*' Or Like '*' & Chr(13) & '*'), validation message: Do not start or end with a blank and do not include double blanks or new line characters!*	-	-
AuthorsAddress	Text	An address applying to the Authors and Copyright statements. *Validation rule: Is Null Or Not (Like ' *' Or Like '* ' Or Like '* *' Or Like '*' & Chr(10) & '*' Or Like '*' & Chr(13) & '*'), validation message: Do not start or end with a blank and do not include double blanks or new line characters!*	-	-
AuthorsURL	Text	A URL web address applying to the Authors or the database content. *Validation rule: Is Null Or Not (Like ' *' Or Like '* ' Or Like '* *' Or Like '*' & Chr(10) & '*' Or Like '*' & Chr(13) & '*'), validation message: Do not start or end with a blank*	-	-

Attributes and indices of the entity 'WorkbenchInterfaceDescription'			
Name	**Type**	**Description / Default value & validation**	**Rqrd. / Index**
		and do not include double blanks or new line characters!	
Index name:		***Attributes & index properties***	
PrimaryKey:		Type *(Primary key; Unique values)*	

WorkbenchInterfaceCompatibility

Documentation of version compatibility between the current module and other modules. Must be manually filled after appropriate testing.

Attributes and indices of the entity 'WorkbenchInterfaceCompatibility'				
Name	**Type**	**Description / Default value & validation**	**Rqrd. / Index**	
ComponentName	Text	The name of a component that is used together with the current component.	R	I (U)
ComponentSignature	Long	A unique number defining a component of the DiversityWorkbench or compatible applications.	R	I(PM)
VersionMajor	Integer	The major number of the version (before the period). *Default value: 0*	R	I(PM)
VersionMinor	Integer	The minor number of the version (after the period). *Default value: 0*	R	I(PM)
ComponentIsCompatible	Boolean	True if the application module is compatible, False if incompatible. *Default value: True*	R	-
ComponentIsRequired	Boolean	False if the module may be missing, True if presence of module is required. *Default value: True*	R	-
TestResponsible	Text	The name of the developer who tested compatibility between module containing this table and the module with ModuleName.	R	-
TestDate	Date/Time	The date when compatibility was tested. *Default value: Now()*	R	-
Index name:		***Attributes & index properties***		
ComponentName:		ComponentName *(Unique values)*		
PrimaryKey:		ComponentSignature; VersionMajor; VersionMinor *(Primary key; Unique values)*		

Footnotes: The following abbreviations have been used in the tables: **R:** It is required to enter data in this field. **I:** The field is indexed to enable faster searching. Different types of indices are denoted by additional letters in parentheses: **P** = attribute is part of the primary key (values in the index must be unique), **U** = values in the index must be unique, **N** = Null values are ignored in the index, **M** = the index contains more than one attribute, **+** = the attribute is involved in more than one multiple-field index. The names of attributes that are part of the primary key are **underlined** in the first column.

Attribute names of system fields and logging fields (record creation or updating) are enclosed in square brackets: [].

Data types: **'Text (255)'** indicates a text of varying length for which no specific design restrictions have been formulated. 255 characters should be read as a proposed technical maximum limit, that can be changed if required by the database management system. In contrast, **'Memo'** is explicitly defined as text of unlimited length. All text is Unicode text. Numeric types: **BigInt** = 8 byte integer, **Long** = 4 byte integer, **Integer** or **Int** = 2 byte integer, **Byte** = 1 byte integer. A **+** (e.g. **Byte+**) indicates that the integer is unsigned and only positive values are allowed.

Footnotes: The following abbreviations have been used in the tables: **R:** It is required to enter data in this field. **I:** The field is indexed to enable faster searching. Different types of indices are denoted by additional letters in parentheses: **P** = attribute is part of the primary key (values in the index must be unique), **U** = values in the index must be unique, **N** = Null values are ignored in the index, **M** = the index contains more than one attribute, **+** = the attribute is involved in more than one multiple-field index. The names of attributes that are part of the primary key are **underlined** in the first column.

Attribute names of system fields and logging fields (record creation or updating) are enclosed in square brackets: [].

Data types: **'Text (255)'** indicates a text of varying length for which no specific design restrictions have been formulated. 255 characters should be read as a proposed technical maximum limit, that can be changed if required by the database management system. In contrast, **'Memo'** is explicitly defined as text of unlimited length. All text is Unicode text. Numeric types: **BigInt** = 8 byte integer, **Long** = 4 byte integer, **Integer** or **Int** = 2 byte integer, **Byte** = 1 byte integer. A **+** (e.g. **Byte+**) indicates that the integer is unsigned and only positive values are allowed.

The following SQL code creates these tables:

```
/*=== Table: WorkbenchInterfaceDescription ===*/
/* This entity contains two records, describing the title, version number, and description of
schema (= information model) and content (= data collection) of a database component
(Workbench Framework 1.0). */
/*     Type: Only two records possible: 'Content' (defined by the user) and 'Schema' (=
information model, defined by developer, must be compatible with the application). */
/*   MajorVersion: The major version number of the schema or content data collection. Example:
'2' for version 2.1. */
/*   MinorVersion: The minor version number of the schema or content data collection. Example:
'1' for version 2.1. */
/*   Revision: The revision version number of the schema or content data collection. Example:
'5' for version 2.1.0005. Displayed only if > 0. */
/*   VersionSuffix: An optional string to identify special versions. Examples: 'beta', 'rc' =
release candidate, etc. */
/*   Signature: A number that uniquely identifies a schema or content data collection (used by
other components to identify object identifiers as belonging to this data collection; content
signature values must be 1-524287, schema 1-4095). */
/*     Title: A title for the schema or content data collection. Example: 'Mycological
Literature collected by the Mycology.Net initiative'. */
/*   Description: An optional description of the entire information schema or data collection
(will be displayed in the 'About' dialog box). */
/*   CopyrightStatement: A copyright statement concerning the data. */
/*   PublicLicense: Type of public licence (none, GPL, LGPL, etc.). */
/*   Authors: A list of author(s). */
/*   AuthorsAddress: An address applying to the Authors and Copyright statements. */
/*   AuthorsURL: A URL web address applying to the Authors or the database content. */
CREATE TABLE WorkbenchInterfaceDescription (
  Type NATIONAL CHARACTER VARYING(7) NOT NULL PRIMARY KEY DEFAULT 'Content',
  MajorVersion INTEGER NOT NULL DEFAULT 1,
  MinorVersion INTEGER NOT NULL DEFAULT 0,
  Revision INTEGER NOT NULL DEFAULT 0,
  VersionSuffix NATIONAL CHARACTER VARYING(20) NULL,
  Signature INTEGER NOT NULL DEFAULT CLng(Rnd()*524287),
  Title NATIONAL CHARACTER VARYING(255) NOT NULL,
  Description NATIONAL TEXT NULL,
  CopyrightStatement NATIONAL CHARACTER VARYING(255) NULL,
  PublicLicense NATIONAL CHARACTER VARYING(20) NULL DEFAULT 'GPL',
  Authors NATIONAL CHARACTER VARYING(255) NULL,
  AuthorsAddress NATIONAL CHARACTER VARYING(255) NULL,
  AuthorsURL NATIONAL CHARACTER VARYING(255) NULL
);
/*=== Table: WorkbenchInterfaceCompatibility ===*/
/* Documentation of version compatibility between the current module and other modules. Must
be manually filled after appropriate testing. */
/*   ComponentName: The name of a component that is used together with the current component.
*/
/*    ComponentSignature: A unique number defining a component of the DiversityWorkbench or
compatible applications. */
/*   VersionMajor: The major number of the version (before the period). */
/*   VersionMinor: The minor number of the version (after the period). */
/*      ComponentIsCompatible: True if the application module is compatible, False if
incompatible. */
/*    ComponentIsRequired: False if the module may be missing, True if presence of module is
required. */
/*     TestResponsible: The name of the developer who tested compatibility between module
containing this table and the module with ModuleName. */
/*   TestDate: The date when compatibility was tested. */
CREATE TABLE WorkbenchInterfaceCompatibility (
  ComponentName NATIONAL CHARACTER VARYING(64) NOT NULL UNIQUE,
  ComponentSignature INTEGER NOT NULL,
  VersionMajor SMALLINT NOT NULL DEFAULT 0,
  VersionMinor SMALLINT NOT NULL DEFAULT 0,
  ComponentIsCompatible BIT NOT NULL DEFAULT 1,
  ComponentIsRequired BIT NOT NULL DEFAULT 1,
  TestResponsible NATIONAL CHARACTER VARYING(255) NOT NULL,
  TestDate DATETIME NOT NULL DEFAULT current_timestamp,
  PRIMARY KEY (ComponentSignature,VersionMajor,VersionMinor)
);
```