Delft Cluster

Title:	**DC-OMS Architecture**

Author:	Ir. S. Hummel	Institute:	WL | Delft Hydraulics
Author:	B.S.T.I.M. The	Institute:	GeoDelft

June 2003

Number of pages	:	26	
Keywords (3-5)	:	Software architecture, data exchange, XML	
DC-Publication-number	:	-	
Institute Publication-number (optional)	:		
Report Type	:	☐	Intermediary report or study
	:	☒	Final project report
DUP-publication Type	:	☐	DUP Standard
		☐	DUP-Science

Other Research project sponsor(s):

Abstract

The aim of the DC-OMS-Architecture project has been to:

- Specify a *DC-OMS-Architecture*,
  in which software-components can perform interaction and data exchange.
  This architecture will describe a set of conventions on how software components exchange their data.
- Realise an *Input/Output-library* (DelftIO),
  based on which components can exchange data, both in memory and by means of files. File exchange is supported for partner-specific file formats, as well as for national (GEF) and international standard XML).

Initially, the Architecture project made an inventory on the various run time and implementation environments that are being used by the project partners. Due to the great variety of supported platforms and programming languages (between partners and even at one partner), the project team concluded that data exchange between software components should be based on self-defining XML-files, unless the concerned data is voluminous (in which case binary file formats and/or on line data exchange should be used).

To understand which types of data will be exchanged in which computation phases, some pilots were performed, which showed that data exchange should be supported for:

- Data blocks (e.g. a 2D-array with the values of a set of quantities on a set of locations where these values are computed).
- Hierarchical data (e.g. part of, or a whole schematisation for a certain model).

To facilitate this data exchange, a software component developer should be provide with a powerful, yet simple mechanism to translate his component specific internal data types into a generic representation, which can be easily stored and/or be retrieved from file.
During the project, this *two-layer approach* (generic layer and component specific layer) has been studied and designed, leading to implementations for

- Fortran 90 / Fortran 77 / Visual Basic / Java / Delphi (data blocks).
- Delphi / Java (hierarchical data)

PROJECT NAME:	**DC-OMS Architecture**	PROJECT CODE:	**07.05.04**
BASEPROJECT NAME:	**DC-OMS (DC Open Modelling Systems)**	BASEPROJECT CODE:	**07.05**
THEME NAME:	**Knowledge Management**	THEME CODE:	**07**

Executive Summary

The Dc-Oms-Architecture project is part of the Delft Cluster base project 'DC Open Modelling Systems'. This base project focuses on coupling existing and/or new software-components that (usually) perform a numerical task.

Initially effort has been put on defining a detailed DC-OMS-Architecture that specifies:

- The exact behaviour of a software-components;
- The implementation environments to be supported.

However, during the project the insight emerged that, for facilitating component coupling, it suffices to specify the mechanism for *data exchange*.

The data exchange between software-components has been specified by means of:

a) Description of the file format to be used;
b) Specification of a two-layer approach for the implementation of data exchange:

The two-layer approach can be summarized as:

1. implement a generic data type (the aggregate or container class), that offers functions to put/get (write/read) aggregated data.
2. based on this aggregate class, implement functions to put/get (write/read) component-specific data types.

Although this approach has not yet been used for exchange of data between components of different companies, it has been used intensively inside companies, and has proven to be very useful and to be well fit for combining the software components of different partners.

PROJECT NAME:	**DC-OMS Architecture**	PROJECT CODE:	**07.05.04**
BASEPROJECT NAME:	**DC-OMS (DC Open Modelling Systems)**	BASEPROJECT CODE:	**07.05**
THEME NAME:	**Knowledge Management**	THEME CODE:	**07**

Applicability for the sector

Software systems are becoming increasingly important for knowledge transfer to the sector; this project contributes to the availability of reusable software components that can be easily integrated into existing software systems.

The baseproject *Delft Cluster Open Modelling Systems* (DC-OMS), and especially the *DC-OMS Architecture* project, facilitate the integration of software components amongst the Delft Cluster partners and the sector.

PROJECT NAME:	**DC-OMS Architecture**	PROJECT CODE:	**07.05.04**
BASEPROJECT NAME:	**DC-OMS (DC Open Modelling Systems)**	BASEPROJECT CODE:	**07.05**
THEME NAME:	**Knowledge Management**	THEME CODE:	**07**

Societal Relevance of the research

Society increasingly requires an integrated approach to water management.

The baseproject *Delft Cluster Open Modelling Systems* (DC-OMS), and especially the *DC-OMS Architecture* project, facilitate the integration of software components amongst the Delft Cluster partners and the sector.

PROJECT NAME:	**DC-OMS Architecture**	PROJECT CODE:	**07.05.04**
BASEPROJECT NAME:	**DC-OMS (DC Open Modelling Systems)**	BASEPROJECT CODE:	**07.05**
THEME NAME:	**Knowledge Management**	THEME CODE:	**07**

Table of contents

List of Figures

List of Tables

1 Introduction

In Delft Cluster Theme 7 (see [DC-7]), the base project 'Delft Cluster Open Modelling systems' (DC-OMS) is a co-operation between the Delft-Cluster partners GeoDelft, TNO-Bouw, and Delft Hydraulics, which all develop, maintain and sell large software systems.

A description of the DC-OMS base project and its projects can be found in Appendix A and in [DC-OMS]. Central project in DC-OMS is the DC-OMS-Architecture project [DC-OMS-ARCH], which aims to:

- Specify a DC-OMS-Architecture,
  in which software-components can perform interaction and data exchange. This architecture will consist of a set of conventions on implementation environments and on the way components are specified and implemented.
- Realise an Input/Output-library (DelftIO),
  based on which components can exchange data, both in memory and by means of files. These files will be based on partner-specific as well as national and international file format standards, like NEFIS (WL), FILOS (TNO), GEF (GeoDelft and various Dutch companies and institutes), XML, and HDF or NetCDF.

The present document describes the DC-OMS-Architecture and introduces the functionality and design of the DelftIO library. This is done in the following way:

- Chapter 2 introduces components and their communication, which will lead to the functional requirements of the Dc-Oms-Architecture
- Based on these requirements, Chapter 3 will present the DC-OMS-Architecture specification.
- Chapter 4 describes the context in which the Dc-Oms-Architecture, and thus DelftIO, will be used, by summarising similar initiatives, and by providing an overview of the development and run time environments of the involved DC-Partners.
- Finally, Chapter 5 elaborates the functional design of DelftIO, while Chapter 6 to 8 will described the DelftIO libraries that have been developed.

2 Components and their collaboration

2.1 Components

Aim of the DC-OMS-Architecture project is to facilitate two or more components to communicate with each other. For a better understanding of the term component, we provide part of the definition that is extracted from [KRUCHTEN]:

> *A component is a non-trivial, nearly independent, and replaceable part of a system that fulfils a clear function in the context of a well-defined architecture. A component conforms to and provides the physical realisation of a set of interfaces. The interface is seen as a 'contract' between the component and its environment and can be defined as a collection of operations that is used to specify a service of a component.*

The fact that a component offers 'services' suggests that it is designed and implemented in an object oriented way. However, DC-OMS components, which mainly are legacy systems of the involved partners, often are not designed and implemented this way.
To restructure the systems in such a way that they *do* adhere to the OO paradigm would be too ambitious and too expensive, and therefore is definitely out of the scope of the DC-OMS project.

Therefore, the DC-OMS-Architecture should interpret the *interface* of a component as the set of operations it performs to:

- retrieve input
- deliver output.

'Input' mainly stands for 'input data', but it should be clear that part of the input could describe the actions to be performed (in which order) by the component, the so-called *control flow*.

2.2 Data exchange

A component usually performs a well-defined task. A component is provided with input data from file or from another component, executes its task, and produces output data, which in turn may serve as input data for another component.
These data flows between two components will often be two-sided (see Figure 1). Component A may be the sender as well as the receiver of data to and/or from component B.

Whatever the mechanism is that is used for the actual exchange of the data, both components need to be aware of the meaning of the data structures that they receive and/or send.
In the ideal situation both components know the data structures to be exchanged, as pictured in Figure 1.

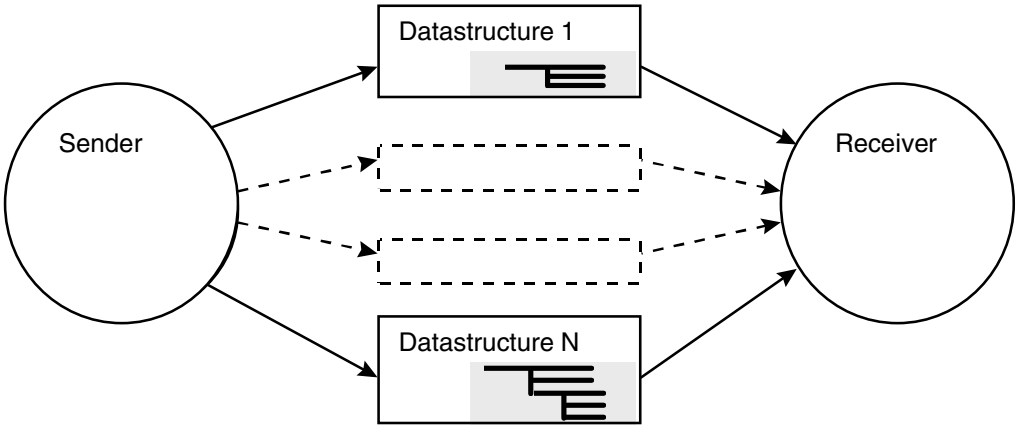

Figure 1, Data exchange between components

However, in the case of existing components, there will often be a discrepancy in the way the data is handled by different components. In this case the following alternatives arise to solve this problem:

- Rewrite both components in such a way that they will use the same data structures
- Rewrite one of the components in such a way that it will use the data structures of the other component
- Create (a) new component(s) that convert(s) the data structures.

The first two alternatives lead to the situation giving in Figure 1 above, while the latter one leads to the situation in Figure 2 below.

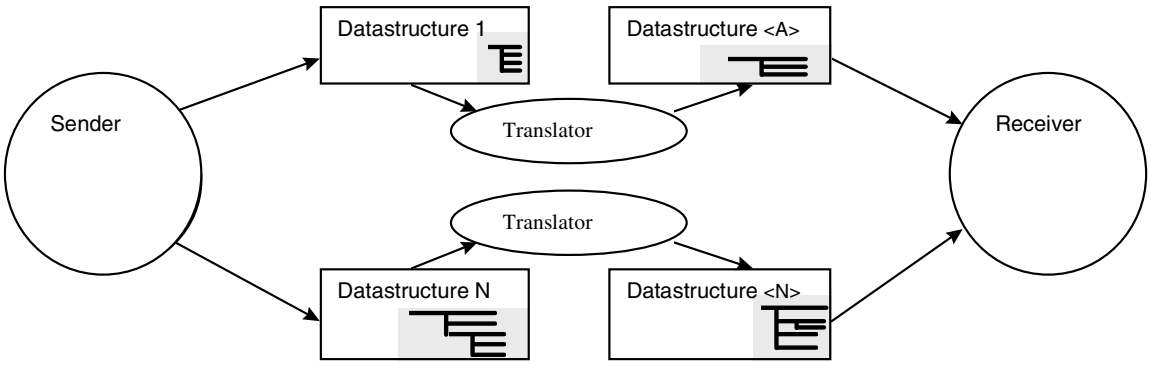

Figure 2, Data exchange between components by means of a translator

It should be clear that the previous paragraphs did *not* mention the way the actual data exchange is realised. Indeed, this implementation aspect is not a part of the architecture specification.
The important issue is that the 'Get/Read' (input) and 'Put/Write' (output) operations of the components are based on well-specified data structures.

The actual data exchange may be implemented by means of file, but also by other mechanisms, like shared memory or CORBA on Windows and UNIX platforms, or COM technology on Windows platforms.

2.3 Data types

To determine which types of data will be exchanged between components, some pilot studies were performed. These pilots are described in detail in [ARCH-PILOTS] (Dutch). Their findings are summarised in the subsections below.

2.3.1 Data exchange MSettle/DIANA

MSettle is GeoDelft's application for computations of vertical stresses and displacements. Although it's powerful 1D-vertical analysis is sufficient for many problem areas, there are cases in which information on the horizontal stresses and displacements would provide additional insight into the physical phenomena involved. Therefore it was decided to use some modules of DIANA for the 2D computation of stresses and displacements (DIANA is TNO-Bouw's application for 2D and 3D mechanics).
To realise this, the data exchange between MSettle and DIANA was analysed and implemented (initially by defining the contents of two files, from MSettle to DIANA and vice versa). A summary of this data is given in Table 1, where some of the data has been elaborated in some detail, to make clear that:

- the data is organised in a hierarchical way
- the data is often not 'one to one' available.

The latter finding once again shows the need for conversion modules, as mentioned in Section 2.2. The first finding imposes a requirement on the DC-OMS-Architecture: it should be able to handle hierarchical data.

MSettle sends	**DIANA receives**

• Geometry:
o Points
o Curves
o Boundaries
o Piezo Lines
o Phreatic Line
o Layers
o Verticals
	• *Mesh (= set of elements)*
	`Point = X: double`
	`Y: double`
	`Z: double`
	`Node = Identification: string`
	`Point: Point`
	`Element = Identification: string`
	`NodeIdentifications[1..n]:string`
	`PhysicalPropertiesIdentification: string`

• Material	• Material properties
	`PhysicalProperties = Identification: string`
	`ElementType: string`
	`CompressieIndex: double`
	`ZwellingsIndex: double`
	`PoissonRatio: double`
	`WrijvingsHoek: double`

• Load: o Non-uniform Loads o Water Loads o Other Loads	• Boundary conditions
DIANA sends	**MSettle receives**
• Stress	• Stress
• Displacement	• Displacement

Table 1, Communication Diana / MSettle

2.3.2 Data exchange Generic Framework models

A Dutch initiative that is comparable to DC-OMS is the Generic Framework (see [GF-ARCH], [GF-TD]). This project focuses on the time step based exchange of quantities on different locations on a computational domain. Two of the models that will be incorporated in this framework are SOBEK (1D-flow) and DELWAQ (1/2/3D-water quality). The data that can be exchanged is given in Table 2.

SOBEK can send	**SOBEK can receive**
For each time step: • Water levels on <NGrid> grid points • Water volumes in <NGrid-1> grid cells • Water levels on <NBound> boundary locations • Discharges on <NBound> boundary locations	For each time step: • List of BoundaryItems, where • BoundaryItem = <water level on boundary location(n)> | • <discharge on boundary location(n)>
DELWAQ can send	**DELWAQ can receive**
	For each time step: Water volumes in <Nseg> segments
For each time step: • Concentration values of <MSubst> substances on <NLoc> locations	For each time step: • List of ConcentrationItems, where ConcentrationItem = <substance(m), value on location(n)>

Table 2, Communication SOBEK / DELWAQ

Once again it is clear that:
- the data is organised in a hierarchical way (less deep however then in the previous section)
- it's important to receive 'lists' of items

- the data is often not 'one to one' available
  (conversion of water volumes on flow grid cells to volumes on water quality segments)

2.4 Requirements for the DC-OMS-Architecture

Given the types of applications that form the DC-OMS components, and given the type of data they exchange, the following requirements can be imposed on the specification of the DC-OMS-Architecture:

1. The DC-OMS-Architecture is mainly meant to be used by the (modules of the) existing legacy systems of the involved partners, and should therefore focus on the way these components receive their input data and send their output data.
   In other words, the DC-OMS-Architecture should be expressed in terms of:
   - *Send* or *Put* or *Write* operations, and
   - *Receive* or *Get* or *Read* operations
   Note: the three names per operation can be regarded as synonyms; they are mentioned explicitly because all these terms are frequently used for data exchange.
2. Because of the wide variety of data used by the systems and modules of the DC-partners, the DC-OMS-Architecture should be able to handle this variety of data, i.e.:
   - It should support the data exchange specification for a *limited* set of 'primitives'. Primitives are the basic scalar types (real, double, integer, Boolean), n-dimensional arrays of these types, and (n-dimensional arrays) of strings
   - It should support the data exchange specification of in fact an *unlimited* set of aggregates of primitives, where an aggregate itself once again is treated as an additional primitive.

3 DC-OMS-Architecture specification

As presented in the previous chapter, the DC-OMS-Architecture describes the way a component interacts with other components, in other words:
- what input is required (and optionally: when)
- what output is delivered (and optionally: when)

So for every component a 'specification' is required in terms of these data exchange operations. Due to the restricted project budget (main effort has been put in developing the DelftIO libraries), this specification method has not been elaborated in detail yet. The following subsections provide a suggestion for these specifications. It should be emphasized that current developments in the water related sector start to focus on formalizing the specification; the Delft Cluster partners should keep track of the progress in these projects, which are:
- HarmonIT (see ref. *HarmonIT*)
- Dutch Oms (see ref. *Dutch Oms*).

3.1 Component specification

A 'component specification' describes a component's functionality and its Get/Put behaviour, as presented in Table 3.

Component	Some Component		
Functionality	Description of functionality		
Data exchange:			
Phase	*Get*	*Put*	*Remarks*
Initialisation	... (e.g. InitialInput)	...	...
Begin of time step	...	...	...
Begin of iteration	...	...	...
End of iteration	...	...	...
End of time step	... (e.g. FlowResult)	...	...
... (other logical phase)	...	...	...
Termination	... (e.g. FlowResult)	...	...

Table 3, Component specification

The data items given in the Put and Get columns are explicitly described in a formal way, the 'data specification'.

3.2 Data specification

The data specification can be described in any suitable way, but preferably is provided in a Pascal-like syntax, for instance:

Point = X: double
Y: double
Z: double

Node = Identification: string
Point: Point

At present, no additional requirements have been imposed on the data specifications. However, in the near future extensions are foreseen regarding meaning, units, and validity of the data items.

4 Dc-Oms-Architecture and DelftIO context

As mentioned in the introduction, a DelftIO library will be realised that supports the exchange of data between the various DC-OMS components. To implement such a library in an optimal way, it should be clear in which operational and development environments it will be used. Also, the DC-OMS approach for integrating components should take into account that there are quite a few similar 'open modelling' initiatives ongoing in the outer world.
Therefore, in an early stage of the project an inventory was made on the context for the DC-OMS-Architecture and DelftIO. This context is described in Appendix 1, as well as the consequences for DelftIO. However, as can be concluded from Table 9 in this Appendix, these consequences all fit into the requirements stated in Section 2.4.

5 DelftIO design and implementation

The main goal of the DC-OMS-Architecture project is to develop a standardised Input/Output library, that takes care of the data-exchange between applications, and that supports self-descriptive data storage, based on national and international file format standards.
However, this library should also support older proprietary formats, so backward compatibility for existing applications can be maintained.

To develop the DelftIO library, some pilots were defined to analyse the types of data to be handled, and the storage of these data types in file formats like XML was investigated.
Based on these experiences we decided that the optimal design for the desired functionality is a two-layer approach:

- The general DelftIO Aggregate/Container/Collection layer.
  This layer provides some basic data object types, including the I/O functions for these general objects, for the various file formats to be supported.
  The basic data object types can be pictured as aggregate and/or collections of data items, like:
  - reals, integers, strings
  - arrays of these primitive elements
  - once again basic data object types.
- The Application Object Specific layer.
  A very thin layer providing the read and write functions for each actual aggregated data type to be supported in typical civil engineering applications, like hydrodynamics, construction mechanics and geotechnical applications.
  This layer converts the application-specific data structure to a basic data object, thus providing its storage in each of the supported file formats.

Details of this approach have been presented in *DelftIO, Standardized Application Communication* (see [DELFTIO-HIC2000]). The approach offers optimal library development and maintenance, since the supported file types as well as the number of specific data types can be extended independently.

5.1 The general DelftIO Data Object layer.

The general DelftIO Data Object is implemented as a collection of elements, and thus and be depicted as an aggregate of data, or as a collection. Such a data object can contain the following elements:

- Integer
- Float
- Double
- Boolean
- String
- N-dimensional arrays of one of the 5 items above
- An other data object
- An array of data objects

Every basic element provides its own (usually very simple) write/read operation to/from different file types. On top of that, the Read/Write operations DelftIO Data Object:

- Perform read/write actions that indicate the beginning of the data object
- Invoke the subsequent read/write operations of the contained elements
- Perform read/write actions that indicate the end of the object

The type of Input / Output stream that the data object is connected to (usually XML), determines which specific read/write implementation will be called.

5.2 The Application Object Specific layer.

Given an existing data-structure in an existing application, a ‘DelftIO-version’ of this data-structure can easily be derived, by using the general DelftIO Data Object.
This DelftIO-version of the application specific data structure effectively should provide two functions, a Read and a Write operation. These functions can simply be implemented by:

- Creating a DelftIO Data Object
- Adding all elements of the existing data structure to this object
- Setting the values for the Write operation

- Invoking the DelftIO Data Object Read or Write function
- Getting the values for the Read operation
- Deleting the DelftIO Data Object.

6 DelftIO for hierarchical data, Delphi

6.1 Functionality

According to the approach described in the previous Chapter, a Delphi DelftIO library has been made available that implements both the general and the application specific layer. This Delphi implementation defines two classes:

- TDIOAggregateDefinition,
- TDIOContainer,

and defines constants to designate the supported primitive data types:

```
const
  sDIOString = 'String';
  sDIOBoolean = 'Boolean';
  sDIOInteger = 'Integer';
  sDIODouble = 'Double';
  sDIOAggregate = 'Aggregate';
  sDIOCollection = 'Collection';
  sDIOArrayDouble = 'ArrayDouble';
  sDIOArrayInteger = 'ArrayInteger';
  sDIOBinaryData = 'BinaryData';
  sDIOReferenceInternal = 'ReferenceInternal';
  sDIOReferenceExternal = 'ReferenceExternal';
```

A TDIOAggregateDefinition objects describes the elements of a certain Container (which is a synonym for Aggregate). Subsequently, a TDIOContainer object is an instance of an aggregate that adheres to this definition.

6.2 Usage

To create a ‘DelftIO-version’ of the application specific data structure ‘Layer’ (which represents a soil layer), the following steps need to be performed:

- Define the aggregate elements and their types, and assign them to a Container:

```
  FDIOLayer: TDIOContainer;
  LLayerAggregateDefinition: TDIOAggregateDefinition;

  LLayerAggregateDefinition.AddElement(
                    sLayerBottomLocation, sDIODouble, '%6.2f', 0.0);
  LLayerAggregateDefinition.AddElement(
                    sLayerTopLocation, sDIODouble, '%6.2f', 0.0);
  FDIOLayer.AddAggregateDefinition(
                    LLayerAggregateDefinition);
```

- Initialize the Container (i.e. build the internal data-storage, based on the definition of the specified AggregateDefinition):

```
FDIOLayer.Initialize(sMyLayer, sLayerIdentifier);
```

- The elements of the container now an be accessed by means of Set/Get functions:

```
procedure SetValue(AIdentifier: string; AValue: string); overload;
procedure SetValue(AIdentifier: string; AValue: Double); overload;
procedure SetValue(AIdentifier: string; AValue: Integer); overload;
function GetValueInt(AIdentifier: string): Integer;
function GetValueDouble(AIdentifier: string): Double;
function GetValueString(AIdentifier: string): string;
```

where `AIdentifier` contains a full path description, for example:

```
LBottom := FDIOBoring.GetValueDouble ('MyBoring.Layers.3.Bottom')
FDIOBoring.SetValue('MyBoring.Layers.3.Top', LTop);
```

6.3 Interface

Table 4 (the Container) and Table 5 (the AggregateDefinition) give an overview of the Delphi DelftIO interface. Full interface documentation is available as HTML-pages at GeoDelft.

TDIOContainer, Properties:	
AggregateDefinitions	Array of aggregate definitions (TDIOAggregateDefinition)
FileFormat	File format: CffUnKnown, CffXML, CffGEF, CffMyformat
GEFMapping	object for translation of GEF format (TDIOGEFMapping)
TDIOContainer, Methods:	
Initialize	Initialize data structure
AddAggregateDefinition	Add an aggregate definition to the definition list
LoadFromFile	Load data from file in selected FileFormat
SaveToFile	Save data to file in selected FileFormat
LoadFromFileGEF	Load datastructure from file (GEF format)
SaveToFileGEF	Save datastructure to file (GEF Format)
LoadFromFileXML	Load datastructure from file (internal XML-datastructure)
SaveToFileXML	Save datastructure to file (internal XML-datastructure)
SetCollectionCount	Set number of items in a collection element in this container
SetValueString	Set a string value of an element in this container
SetValueDouble	Set a double value of an element in this container
SetValueInt	Set an integer value of an element in this container
SetValue	Set a value of an element in this container
GetCollectionCount	Get number of items in a collection element in this container
GetValueInt	Get an integer value of an element in this container
GetValueDouble	Get a double value of an element in this container
GetValueString	Get a string value of an element in this container
HandleSetValueString	Event handler to set a string value of an element in this container
HandleSetValueDouble	Event handler to set a double value of an element in this container
HandleSetValueInt	Event handler to set an integer value of an element in this container
HandleGetValueInt	Event handler to get an integer value of an element in this container
HandleGetValueDouble	Event handler to get a double value of an element in this container
HandleGetValueString	Event handler to get a string value of an element in this container

HandleSetCollectionCount	Event handler to set number of items in a collection element in this container
HandleGetCollectionCount	Event handler to get number of items in a collection element in this container

Table 4, TDIOContainer interface

TDIOAggregateDefinition, Properties:	
Identifier	Unique identifier of this instance
ElementCount	Number of elements in this definition
Elements	Array of elements (TDIODataElementType)
ElementByString	Array of elements indexed by identifier (TDIODataElementType)
TDIOAggregateDefinition, Methods:	
AddElement	Add aggregate element
FindElementIndex	Find index of element by identifier

Table 5, TDIOAggregateDefinition interface

6.4 DIO-Library GEF Implementation

GeoDelft has used the implementation of the Delphi DelftIO library to create a component to read and write a GEF-Boring file. A GEF-Boring file is a recently defined standard to exchange borings.
GEF is a data exchange file format widely used in the geotechnical world. The most well known GEF format is the GEF-CPT (see refs. [GEF], [GEF-CPT], [GEF-BORE]).

For the GEF implementation we define two objects:
- TDIOGEFMapping: this is the object in which the specific GEF format is defined (in this case the GEF-Boring)
- TDIOGEFReader: this is the object which is responsible for actual reading and writing the GEF-File, based on the definition of the TDIOGEFMapping

To implement the GEF-Boring reader the following steps are needed:
1. Define the object inherited from TDIOContainer to define the Boring. In this case TDIOBoring.
2. Create an object that is inherited from TDIOGEFMapping, e.g. TDIOGEFMappingBoring. Define the mappings of the GEF data-elements to the DIO object elements:

```
{ GEF Header mappings }
AddElementMapping(sBoringIDEndDepth, geMeasurementVar,
                  CMeasurementVarEndDepth);
{ Scandata mappings }
AddElementMapping(sBoringIDLayerCount, geLastScan, CNoCode);
AddElementMapping(sBoringIDLayers, geScandata, CNoCode);
AddElementMapping(sLayerTopLocation, geColumnInfo, CColumnInfoTop);
AddElementMapping(sLayerBottomLocation, geColumnInfo, CColumnInfoBottom);
```

Reading a GEF file is the accomplished by the following code:

```
LGefMappingBoring := TDIOGefMappingBoring.Create;
try
  FDIOBoring.FileFormat := FileFormat;
  FDIOBoring.GefMapping := LGefMappingBoring;
  FDIOBoring.LoadFromFile(sMyBoring);
finally
  LGefMappingBoring.Free;
end;
```

7 DelftIO for hierarchical data, Java

7.1 Functionality

According to the approach described in the previous Chapter, a Java package (the 'Aggregate package) been made available that implements the general data object layer (see Section 5.1). In fact, this package offers the same functionality as the TDIOContainer class mentioned in the previous Chapter. The Aggregates can be created, accessed, and can written/read to/from XML.

Full documentation on the Java Aggregates (functionality, examples of usage, and interface specification) is available in reference *Aggregate Design*. In the present report, we just summarize the interface (see Table Table 6):

Class	*Property/Method*	*Description*
Aggregate	Key	Unique identification in persistent format
	Name	User identification
	Type	Type of aggregate
	FileOwner	Denotes whether child files are to be copied or deleted if requested
	Deleted	Specifies that the aggregate will be removed when the aggregate is written in the database.
	Parent	The parent of the aggregate
	Database	The database to be used for saving and restoring
	Attributes	Internal array of Relation objects
	GetValue	Gets a child object identified by an attribute or a list index
	SetValue	Sets a child object identified by an attribute or a list index. This can mean an insert or an update of a child object.
	AddValue	Adds an aggregate as a child object. An attribute is generated by first taking the type of the aggregate and then appending a number, in such a way that a unique attribute is created/
	RemoveValue	Removes an attribute
	GetCount	Returns the number of attributes
	GetIndex	Gets the list index of the specified attribute
	SetIndex	Sets the specified attribute at the specified position in the list.
	GetAttribute	Gets the attribute at the specified list index.
	CopyFrom	Copies all child objects from a given source aggregate to this aggregate. An ancestor is given, which specified when to create a new instance of the child object (if the child object has the given ancestor as one of it's ancestors) or when to create a reference to the child object (otherwise)
Relation	Attribute	The attribute string
	Child	The child object
	Owner	The aggregate containing the child, mostly equal to the parent of the child.
Database	Read	Abstract definition of the restore method
	Write	Abstract definition of the save method

Class	*Property/Method*	*Description*
SQLDatabase	Read	Implementation of the restore method for an SQL database
	Write	Implementation of the save method for an SQL database
XMLDatabase	Read	Implementation of the restore method for an XML database
	Write	Implementation of the save method for an XML database
	Import	Reads a specified file and returns the aggregate saved in it.
	Export	Writes an aggregate to a specified file
	SetRoot	Defines that aggregates of the specified type will be the root of an XML file
	IsRoot	Tells whether an aggregates of the specified type is the root of an XML file

Table 6, Java Aggregates interface

8 DelftIO for data blocks

8.1 Functionality

The Delft IO library for data blocks in fact only implements the application-specific layer (see Section 5.2). The reason for that is that this version of the library is dedicated to a data type that is widely used in the components of Delft Hydraulics' SOBEK-applications. This data type contains ***P**arameter/**L**ocation* values per **T**ime step, and therefore is known as the PLT type. Figure 3 pictures the contents of a PLT dataset.

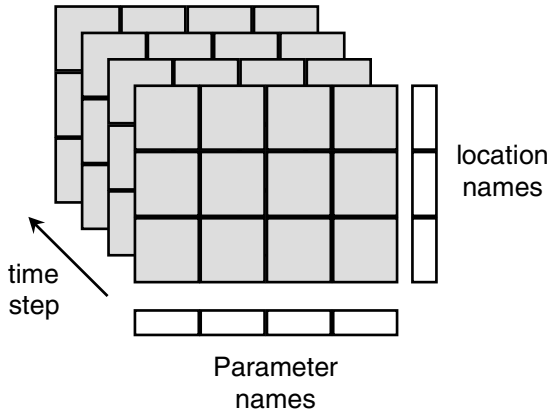

Figure 3, Parameter / Location / Time step dataset

DelftIO-PLT offers a variety of functions for defining (i.e. creating and writing) and getting (i.e. reading) these PLT datasets.
The PLTs can be exchanged by file (currently only in Delft Hydraulics's internal formats), and by means of shared memory. A MPI based interprocess version will be available soon.

8.2 Interface

An overview of the Fortran 90 interface is given in Table 7. This Fortran 90 interface is used by the computational core components. For other applications, like:

- Fortran 77 computational cores
- user interface components in SOBEK,
- the model wrappers in the Generic Framework,

interfaces in additional languages (Fortran 77, VB6, Delphi) have been put on top of this Fortran 90 interface.
Full interface documentation is available as HTML-pages at Delft Hydraulics. Detailed information on the design and implementation can be found in *Technical Documentation DelftIO* [DELFTIO].

DioPLT, Type definitions	
type DioPlt	Handle to dataset
DioPLT, functions/subroutine	
DioPltDefine	Define a Par./Loc./Time dataset (overloaded function) The function returns a handle to the PLT dataset.
DioPltPut	Put values for a specific Julian time stamp, for a specific His time step, or for the next time step (for on line communication)
DioPltGetDataset	Get a Par./Loc./Time dataset, optionally from a Stream. The function returns a handle to the PLT dataset
DioPltOpenedOK	Check if a Par./Loc./Time was opened successfully
DioPltGetNPar	Get number of parameters in the PLT dataset
DioPltGetNLoc	Get number of locations in the PLT dataset
DioPltGetNTimes	Get number of time steps in the PLT dataset
DioPltGetPars	Get the parameter names in the PLT dataset
DioPltGetLocs	Get the location names in the PLT dataset
DioPltGetTimes	Get the time steps in the PLT dataset (Julian time stamps)
DioPltGet	Get floats (=reals) for the next time step.
DioPltGetSelection	Get floats (=reals) for a selection of parameters, locations and time steps; or get all parameter/location values for one time step
DioPltRewind	Rewind a (serial) file containing a PLT
DioPltClose	Close and destroy the PLT dataset

Table 7, DelftIO interface for Parameter / Location / Time step datasets

9 References

[DC-7] Price, R.K.P., Mynett, A.E.M., ed.,
Research Theme 7, Knowledge Management, research programme 1999-2002.
Delft Cluster, 1999.

[DC-OMS] Schrepper, G.M.A., Engering, F.P.H., Hummel, S.,
Basisprojectplan DC Open Model Systeem (DC-OMS),
Delft Cluster, 2000.

[DC- OMS-ARCH] Hummel, S.,
DC-OMS Architectuur Projectplan (Dutch),
Delft Cluster, 2000.

[ARCH-PILOTS] The, B.S.T., Hummel, S.,
DC-OMS Architectuur Pilots (Dutch, contact authors at WL|Delft Hydraulics, GeoDelft),
Delft Cluster, 2002.

[DELFTIO-HIC2000] Hummel, S., The, B.S.T., Branchett, S.E., Brinkman, R.,
DelftIO, Standardized Application Communication
in *Proceedings of the 5th International Conference on Hydroinformatics*, pp. 630–637, IWA publishing, London, 2002.

[DUTCH-OMS] Dutch-OMS,
Migrate from SIMONA and Delft3D to one Dutch Open Modelling System (OMS),
http://www.dutch-oms.org (RWS/RIKZ, WL | Delft Hydraulics).

[KRUCHTEN] P. Kruchten,
Modeling Component Systems with UML, (a.o.) in Int. Workshop on Component Based SE, 1998,
http://www.sei.cmu.edu/cbs/icse98/papers/p1.html

[GF-ARCH] Van der Wal, T. (ed.),
Generic Framework Water Architecture,
Generic Framework, The Netherlands, 1999.

[GF-TD] Tacke, J., Brinkman, R., Frieswijk, E., Levelt, D., Otjens, T.,
Generic Framework Technical Design,
Generic Framework, The Netherlands, 2000.

[GEF] Stichting CUR,
GEF, Definition of the GEF language (see GeoNet),
Stichting CUR, Gouda, The Netherlands, 2000.

[GEF-CPT] Stichting CUR,
GEF-CPT-Report, Geotechnical Exchange Format for CPT-Data (see GeoNet),
Stichting CUR, Gouda, The Netherlands, 2002.

[GEF-BORE] Stichting CUR,
GEF-BORE-Report, Geotechnisch uitwisselingsformaat voor Boor-Data (Dutch, see GeoNet),
Stichting CUR, Gouda, The Netherlands, 2002.

[DELFTWISE] Gijsbers, P.J.A., Brinkman, R., Levelt, D.F.,
DelftWISE specificaties (Dutch),
WL|Delft Hydraulics, 2000.

[XML] World Wide Web Consortium,
XML (Extended Markup Language),
W3C.

[HARMONIT] HarmonIT,
Development and implementation of a European Open Modelling Interface (OpenMI),
http://www.harmonit.org

[DELFTIO] Hummel, S.,
Technical Documentation DelftIO, Fortran 90 / Shared Memory / Tests
Delft Hydraulics, 2002.

[DELFTIO-AGGR] Brinkman, R.,
Aggregate Desing,
Delft Hydraulics, 2003.

Appendix 1 Context of the DC-OMS-Architecture and DelftIO

As mentioned in the introduction, a DelftIO library will be realised that supports the exchange of data between the various DC-OMS components. To implement such a library in an optimal way, it should be clear in which operational and development environments it will be used. The following Sections will describe these environments.

Inventory of systems and tools at DC-partners

Before specifying the DelftIO library, an inventory has been made of the environments in which the models of the various DC-partners are developed and run (i.e. the operating systems, programming languages, and additional tools). The results of this inventory are presented in Appendix B. It appears that Windows as well as various UNIX platforms are supported, and that the following languages are in use:

- *C / C++ (for user interfaces and computational parts)*
  On Windows platforms the Microsoft Foundation Classes (MFC) are used, on UNIX the X and OSF-Motif libraries. Also the multi-platform tool XVT is used.
- *Fortran 77 / Fortran 90 (for computational cores)*
- *Java (for user interfaces)*
- *Delphi (mainly for user interfaces)*
  Right now Delphi is only available on Windows. In the 'Kylix' project, Borland is porting Delphi to Linux.
- *Visual Basic (for user interfaces)*
  This environment is only available on Windows.

Effectively this means that, besides of the functional requirements for the DelftIO library from Section 2.4, there is a strong technical requirement imposed on this library: it should support a wide variety of languages and platforms.
In short term, it will be difficult to support all languages and platforms at once. Therefore it has been decided that DelftIO will be developed in different libraries, each supporting the currently most needed languange and data type combination.

Similar initiatives

The DC-OMS-Architecture is related to comparable Dutch and international initiatives, which are listed in Table 8. Details of these initiatives can be found in the References (Chapter 9).

Project	**Partners**	**Focus**	**References**
Dutch OMS (Open Modelling System)	RWS/RIKZ, Delft Hydraulics	Restructuring of modules for 3D hydrodynamics, transport and morphology; Design and implementation of an underlying architecture ('backbone') for data exchange, synchronisation and parallelisation.	*Dutch-OMS*
GF (Generic Framework)	Six Dutch companies and institutes	Time step based exchange of a varying number of quantities on a varying number of locations in several models and/or domains	*GF-ARCH, GF-TD*
GEF-Bore	CUR, GeoDelft	Extension of GEF for storage of bore hole information	*GEF, GEF-BORE*
DIANA-Specials Group	TNO-Bouw	Design and implementation of a generic library for accessing hierarchical data (initially based on Filos, extendible to, for instance, XML).	No external reference available yet

DelftWISE	Delft Hydraulics	Delft Hydraulics' System Architecture, focussing on data exchange, case management (the *DataServer*) and module invocation order and conditions (the *WorkflowServer*).	*DelftWISE specificaties*
HarmonIT	Twelve European companies and institutes	Data exchange between hydrodynamics, transport and other models.	*HarmonIT*

Table 8, Related and comparable initiatives

Given the focus of these projects, Table 9 describes the consequences for the DC-OMS-Architecture and especially for the DelftIO library.

Project	**Consequences for DC-OMS-Architecture**
Dutch OMS	DelftIO must support effective exchange of large <n>D-data-arrays.
GF (Generic Framework)	DelftIO must be implemented in such a way, that one of the exchange mechanisms is the one used in GF
GEF-Bore	DelftIO must be able to read a GEF-Bore file
DIANA-Specials Group	DelftIO must be compatible to the C++-library of the DIANA Specials Group
DelftWISE	DelftIO must be able to attach to the DelftWise DataServer and the DelftWise WorkflowServer
HarmonIT	No implications yet, however: the DelftIO developers should keep track of the progress of and developments in HarmonIT.

Table 9, Consequences for DelftIO

General Appendix: Delft Cluster Research Programme Information

This publication is a result of the Delft Cluster research-program 1999-2002 (ICES-KIS-II), that consists of 7 research themes:
►Soil and structures, ►Risks due to flooding, ►Coast and river , ►Urban infrastructure, ►Subsurface management, ►Integrated water resources management, ►Knowledge management.

This publication is part of:

Research Theme	:	Knowledge Management		
Baseproject name	:	DC-OMS (Delft Cluster Open Modeling Systems)		
Project name	:	DC-OMS Architectuur		
Projectleader/Institute		Ir. S.Hummel		GeoDelft
Project number	:	07.05.04		
Projectduration	:	01-04-2000	-	30-06-2003
Financial sponsor(s)	:	Delft Cluster		
		GeoDelft		
		WL|Delft Hydraulics		
		TNO Construction		
		Stowa (GF, Generic Framework)		
Projectparticipants	:	WL|Delft Hydraulics		
		GeoDelft		
		TNO Construction		
Total Project-budget	:	€ 200.000		
Number of involved PhD-students	:	0		
Number of involved PostDocs	:	0		

Delft Cluster is an open knowledge network of five Delft-based institutes for long-term fundamental strategic research focussed on the sustainable development of densely populated delta areas.

Keverling Buismanweg 4
Postbus 69
2600 AB Delft
The Netherlands

Tel: +31-15-269 37 93
Fax: +31-15-269 37 99
info@delftcluster.nl
www.delftcluster.nl

Theme Managementteam: Knowledge Management

Name	**Organisation**
Prof. R.K. Price	IHE
Prof. Dr. Ir. A.E. Mynett	WL|Delft Hydraulics

Projectgroup

During the execution of the project the research team included:

Name	**Organisation**
Stef Hummel	WL|Delft Hydraulics
Tom The	GeoDelft
Jos Jansen	TNO Construction
Susan Branchett	TNO Construction