UI API Specification documentation

I Basic Types

Basic types are described by just a *`Table`* with a type string inside. For example a `String` is described by this table :

```
[ @type : "String" ]
```

Available basic types are : `String, Boolean, Integer and Void.`

II User-Defined Types

User-defined types are described by a *`Table`*, but the contents vary.

Available user-defined types are : `Struct, List, Vector, Array , Enumeration and Union .`

In-Depth Definitions

Struct

```
[
    @type : "Struct",
    @userDefClassId  : <String>,
    @members : [
        [
            @name : <String>,
            @typeInfo : <basic type> | <user-defined type>
        ],
        …,
        [
            @name : <String>,
            @typeInfo : <basic type> | <user-defined type>
        ]
]
```

Notes :

The field `members` holds a *`Table`* with all the included struct member datatypes. Each name must be unique.

`List (of elementTypeInfo)`

```
[
    @type : "List",
    @userDefClassId : <String>,
    @elementTypeInfo : <basic type> | <user-defined type>
]
```

Vector (of elementTypeInfo)

```
[
    @type : "Vector",
    @userDefClassId : <String>,
    @elementTypeInfo : <basic type> | <user-defined type>
]
```

Array (of elementTypeInfo)

```
[
    @type : "Array",
    @userDefClassId : <String>,
    @elementTypeInfo : <basic type> | <user-defined type>
    @length : <Number>
]
```

Notes :

The field `length` must be an integer greater than zero.

Enumeration

```
[
    @type : "Enumeration",
    @userDefClassId : <String>,
    @members : [
        [ @name : <String> ],
        …,
        [ @name : <String> ]
    ]
]
```

Notes :

The field `members` holds a *Table* with all the possible enumeration values. Each name must be unique.

Union

```
[
     @type : "Union",
     @userDefClassId : <String>,
     @members : [
          [
               @name : <String>,
               @typeInfo : <basic type> | <user-defined type>
          ],
          …,
          [
               @name : <String>,
               @typeInfo : <basic type> | <user-defined type>
          ]
     ]
]
```

Notes :

The field `members` holds a *Table* with all the included datatypes. The union produced will be string discriminated. For that reason each name must be unique.

III Function Specification

Function specifications are described by a *Table*. The two basic fields are the function signature (`signature`) and the reference to the actual function (`func`).

In-Depth Definition

```
[
    @signature : [
        @name : <String>,
        @returnValue : [
            @name : <String>,
            @typeInfo : <basic type> | <user-defined type>
        ],
        @parameters : [
            [
                @name : <String>,
                @dataFlowType : <String>,
                @typeInfo : <basic type> | <user-defined type>
            ],
            …,
            [
                @name : <String>,
                @dataFlowType : <String>,
                @typeInfo : <basic type> | <user-defined type>
            ]
        ]
    ],
    @func : [
        @ref : <ProgramFunc> | <LibFunc>
    ]
]
```

Notes :

The function name (`signature.name`) must be unique. Each parameter name must be unique. `DataFlowType` can be one these values : `"In"`, `"InOut"` and `"Out"` . `"In"` refers to data passed to the function, `"Out"` refers to data coming from the function and `"InOut"` refers data going both ways.

IV UI API Specification

The UI API specification is a *`Table`* containing all the function specifications defined. This is subject to change.

In-Depth Definition

```
[
	<function specification>,
	…,
	<function specification>
]
```

V *Function Implementation Notes*

Data Flow types

function parameters that are defined with an `Out` or `InOut` `dataFlowType` will have their argument data stored inside a *`Table`* in index 0. For example :

```
// Argument a is of InOut dataFlowType.
// Assume that the string "hello" was passed as input.

function f (a) {
	std::print(a[0]); // will print "hello"
	a[0] = "world";   // will set the argument's value as "world"
}
// After the function is invoked, the micro UI showing a's value will
// now show "world".
```

Parameters that are defined with an `In` `dataFlowType` can be used normally.

Basic Type Arguments

Basic type objects map to their respective Delta type. For example a Boolean object is a Delta *`Boolean`* .

Struct Arguments

Struct objects are tables with their defined names as indices and their data as values. For example assume this struct definition :

```
personStructDefinition = [
	@type : "Struct",
	@userDefClassId  : "Person",
	@members : [
			[ @name : "Name", @typeInfo : [ @type : "String" ] ],
			[ @name : "Surname", @typeInfo : [ @type : "String" ]]
	]
];
```

The struct *`Object`* will be like this :

```
person = [
     @Name : <String>,
     @Surname : <String>
];
```

Union Arguments

Union objects are type and value tuples since our unions are string discriminated.
For example, assume this union definition.

```
UnionDefinition = [
     @type : "Union",
     @userDefClassId : "sampleUnion",
     @members : [
          [
               @name : "StringElement",
               @typeInfo : [ @type : "String" ]
          ],
          [
               @name : "IntElement",
               @typeInfo : [ @type : "Integer" ]
          ]
     ]
];
```

The union *`Object`* will be like this :

```
union = [ @type : "StringElement", @value : <String> ]
```

or

```
union = [ @type : "IntElement", @value : <Number> ]
```

Notes :

@type is used for Delta CORBA compatibility.

Array Arguments

Array objects are tables with numerical indices up to length -1.
For example, assume this array definition :

```
strArrayDefinition = [
     @type : "Array"
     @userDefClassId : "StrArray",
     @length : 5,
     @elementTypeInfo : [ @type : "String" ]
];
```

The array *`Object`* will be like this :

```
strArray = [ <String>, <String>, <String>, <String>, <String> ];
```

Vector Arguments

Vector objects are tables with numerical indices.

Notes :

Tables with numerical indices are used instead of *`std::vector`* for Delta CORBA compatibility.

List Arguments

List objects are of type *`std::list`* .

Enumeration Arguments

Enumeration objects are just strings.