# Introduction

This document is a first attempt to describe some specifications for the X# runtime. The document is the result of discussions we had in Athens in December 2016 combined with some of my personal notes from the past year.

Obviously: the plan is to create everything in X# when possible.

The only exception is the Macro Compiler which will have to be in C# because of the C# template we are using for Antlr.

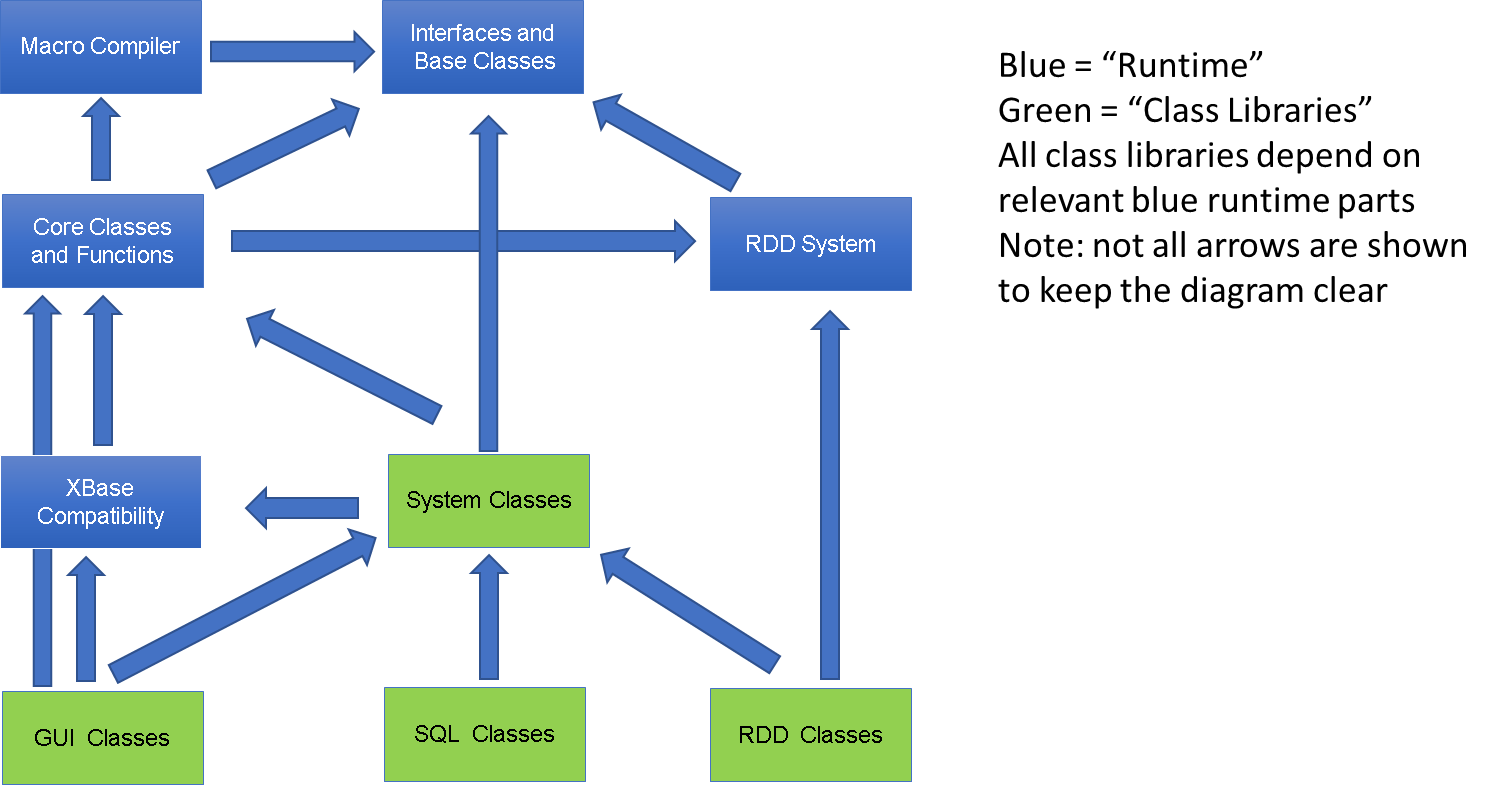
I think all the components should be written in X# Core dialect.

No ‘mysterious’ implicit conversions, implicit references to assemblies etc. Everything should be as clear as possible.

That also means that inside the assemblies we will use the special types by their special name, so for example “\_\_XUsual” instead of USUAL.

Robert

# Assemblies



# Contents of the various assemblies

## Interfaces and Base Classes

This assembly contains classes like:

* Attributes needed by the compiler (similar to what Vulcan does)
* Interfaces for the RDD system, such as IRDD, IDate, IFloat, IArray
* Interfaces for the Macro Compiler

There will be No state in this assembly. And it should only be dependent on System.

Suggested name: **XSharp.Base**

Suggested namespace: **XSharp.Internal**

## Macro Compiler

The macro compiler will only have a dependency on the Interfaces and Base classes. It will implement IMacroCompiler and will return objects that implement ICodeBlock.

There does not have to be a relation between the macro compiler and other parts of the runtime. The macro compiler will use the same mechanism that the normal compiler uses to locate functions. But instead of using a static list of references (passed through the command line) it will use the list of currently loaded assemblies to locate the special Functions classes and find the functions in these classes.

Suggested name: **XSharp.MacroCompiler**

## RDD System

The RDD system will have its own “State”, such as the values for:

* SetDeleted()
* SetCollation()
* SetUniqiue()
* Workarea list
* Current workarea (name and number)

The functions that manipulate that state will be in the Core Classes and Functions library. The RDD system will only save the state itself.

There will be “global’ state for the main thread, and when tables are opened in a thread then a copy of the global state will be made.   
However each thread will have its own workarea list and current workarea

There will be RDDs for:

* DBF
* DBT (builds on top of DBF)
* FPT (builds on top of DBF)
* DBFCDX (builds on top of FPT)
* DBFNTX (build on top of DBT)
* Others, such as NSX/SMT
* Advantage (RDD communicates with ACE32/ACE64, like Advantage RDD for Vulcan)
* Delim, includes proper support for CSV
* SDF

There are quite some variations of DBFs around. I suggest that we add the ability to the DBF class to read and write all known field types. Specific subclasses can indicate through an ‘allowed fieldtypes’ property which fieldtypes should be allowed when creating tables for a specific type of DBF. For example: FoxPro has added AutoIncrement, Integer, Double, TimeStamp. I think it is relatively easy to code all of this in the DBF class themselves. We only have to make sure that when creating DBFNTX these field types should not be included.

The helper classes and structures such as DBLOCKINFO and DBFILTERINFO will also be located in the RDD assembly.

The RDD system will not communicate directly with the Macro compiler. There will be a slot in the global RDD state where the macro compiler will be registered. The RDD system will then be able to use the macro compiler interfaces to compile strings into code blocks.

We need to somehow register the macro compiler in the RDD system at startup.   
That could be triggered by a function as RddSetDefault() or SetDeleted(). Functions like that are called in all standard apps and will force the loading of the RDD system by the Core runtime, and will setup the link between the macro compiler, the Runtime and the RDD system.

The RDD source from the Harbour project can be used as inspiration for writing our RDDs.

There will some differences between the X# RDDs and the RDDs inside VO and Vulcan:

* Each workarea will have its own Deleted and Collation, Ansi, Codepage, Encoder and other relevant state.
* The methods in the RDD method table will be based on the RDD methods found in VO, Vulcan and Harbour.
* The methods in the RDD classes will no longer return values in ByRef parameters but will return the values through return values. Error conditions will not be returned in a logical return value but will throw an exception.  
  This means for example that where Vulcan has an EOF method with the following prototype:  
  VIRTUAL METHOD Eof(lpfStatus REF LOGIC) AS LOGIC

This method will be replaced in our RDD with

VIRTUAL PROPERTY Eof AS LOGIC GET

* The RDD system will use a Factory class to create the Stream objects used to access the file system. The default factory will use the normal System.IO.FileStream class, but you can also register another factory to create other type of streams.
* All codeblock evaluations will go through one method that takes care that the current workarea is set before a codeblock is evaluated and restored afterwards. Since each thread has its own current workarea this should not cause any problems in MT code.
* One of the things that I did not like in the Vulcan implementation of the RDD system is that it was not easy to replace either the index part or the memo part of the RDD. The DBFMEMO class for example adds Flexfile memos to DBFNTX. That was only possible by duplicating a lot of code with the DBFCDX class.  
   I have used conditional compilation in Vulcan to share the same code in 2 different classes and specify 2 different parent classes.

I think it will be easier to do this if we split the object model into:

- DBF Class  
- Index Class   
- Memo Class  
The DBF Class will hold a reference to an object that handles all the index related operations as well as a reference to an object that handles all the memo operations. These objects get a reference back to the DBF object so they can access properties and methods in there.  
With this layout it is much easier to mix implementations: the DBFMEMO Class can still inherit from DBFNTX but then instantiate an a FlexFile compatible Memo class.  
The same memo class is then also used for the DBFCDX driver.

* We may also want to support adding other functionality to the RDD classes, such as Triggers (which Harbour supports), passwords etc.  
  Harbour also has some extra function in the RDD layer (that could be implemented as static methods in the RDD classes) to check if a table Exists, rename a table, Drop a table etc. We may want to do something similar.

Suggested name: **XSharp.RDD**

Suggested namespace: **XSharp.RDD**

## Core Classes and Functions

This assembly will be designed so it can be used from the Core dialect, but will also allow access to the RDD system.

It will contain many of the familiar XBase Functions, such as Left, Right,

Some of the XBase types that are used in the RDD system will also be implemented in this Assembly:

* Codeblock
* Date
* Float

The suggested internal names for the types are \_\_X<Type>, located in the XSharp namespace.

Special care is needed for these types, since there is a lot of logic in the X# compiler when compiling from and to these types. The compiler ‘expects’ that explicit and implicit operators are available for a lot of things.   
The best approach is to copy the operator layout from the Vulcan Runtime and then implement the code manually.

Other types that are in this Assembly are:

* OleAutoObject
* Error Class

This assembly will have VODB() functions like the VO runtime to communicate in a strongly typed manor with the RDD system. The only difference will be that the USUAL parameters will be changed to OBJECT, since the USUAL type is not included in this assembly.

An example is VODBFieldGet(). This function is defined in VO and Vulcan with parameters of type DWORD and USUAL. There will be a function in this assembly that will have parameters of type DWORD and OBJECT.

This assembly will also contain the runtime “State”, and functions to set/get this state such as SetExact(), SetDecimals() etc. There will also be functions to set/get the state from the RDD system.

The runtime state will not be a class with hardcoded fields for specific values, but will be implemented as a dictionary to make it as flexible as possible. Each slot in the dictionary with have a numeric key (like the SET defines in Clipper/VO, such as \_SET\_EXACT) and an object value. Getter/Setter functions will be strongtyped and will cast the object value to the correct type. There may be a small overhead in boxing some values, but the advantage that the dictionary is flexible and does not have to be adjusted outweighs this disadvantage.

The Core runtime should also use a factory class to create streams for file I/O to make the system as portable as possible. The default implementation of this factory class can be based on System.IO or directly use IO functions from Win32

Suggested name: **XSharp.Core**

Suggested namespace: **XSharp**

## XBase Compatibility

This assembly is used for apps that compile in the VO and/or Vulcan dialect.

This assembly will add the special XBase types to the runtime and the functions that use these XBase types.   
The types are

* Array
* PSZ
* Symbol
* Usual

The suggested internal names for the types are \_\_X<Type>, located in the XSharp namespace.

Special care is needed for these types, since there is a lot of logic in the X# compiler when compiling from and to these types. The compiler ‘expects’ that explicit and implicit operators are available for a lot of things.   
The best approach is to copy the operator layout from the Vulcan Runtime and then implement the code manually.

This assembly will have the DB..() functions to communicate with the RDD system. These functions often take USUAL parameters or return USUAL values.

There will also be overloads in this assembly for functions that are defined in the Core runtime with an OBJECT parameter which expect a USUAL parameter in VO and Vulcan, such as VODBFieldGet().

For the VO dialect, we will also need to add support for PUBLICs and PRIVATEs. That will need some adjustments in the runtime, since the visibility of these variables requires special handling, especially since throwing exceptions can unwind a complete huge call stack, and all the privates allocated in this stack need to be cleared.

When the VO Compiler detects usage of PRIVATEs it does the following:

* Each function that defines new PRIVATES has a special function call at the beginning. This records the current stack level on the memvar stack. When the function is finished, all privates allocated later will be cleared from the memvar stack.
* To handle unwinding the privates stack when exceptions are thrown (or the BREAK statement is used) then we need to add code into the RECOVER clause of the BEGIN SEQUENCE .. END , and also the CATCH clause for a TRY statement in which the memvar stack is reset.

This is technically not a big problem, but adding code like this should only be done when privates and/or public are actually used. We may want to add a compiler option to enable/disable this feature.

It is probably a good idea to create 2 different versions of these assemblies: one with VO compatible MEMVARs and one without. These assemblies can share 90% of the code or maybe more.

Suggested name: **XSharp.Vulcan** and **XSharp.VO** for the memvar support.

Suggested namespace: **XSharp**

## SDK Classes

We will start with SDK classes that are recompiled from the AEF files in the VO28\Sdk folder. This has the disadvantage that we will need to make some code changes again that were done for the conversion from VO to Vulcan.

The big advantage is that defines inside these AEFs will be handled like the DEFINES in VO and will become constants in the .Net assemblies. By doing that we will get rid of most of the header files, which will also be good for compile time performance.

And there is also the legal issue: by basing our libraries on the SDK source that is copyright Computer Associates we will not have to deal with potential copyright claims from GrafX.

I suggest we do this:

* Start with the VO 2.8 SDK
* Convert this to our SDK Projects (XIDE and/or VS)
* Make the necessary adjustments to get code that works inside X#

In a later stage, we will replace this code with optimized code. Weakly typed properties, such as HyperLabel.Caption (= USUAL in VO and Vulcan) will be correctly typed.   
This will most likely require a change in the compiler to detect this and ‘automatically’ convert same named untyped properties in subclasses to the new types.

### System Classes

This will be an optimized version of the VO System classes.

### GUI Classes

We will replace the GUI classes library with a library created on top of WinForms that has the same methods and properties that the VO GUI classes have. Most of this I have done already for a customer project.

### SQL Classes

We will replace the SQL classes library with a library created on top of ADO.NET with the same methods and properties from the VO SQL Classes Most of this I have done already for a customer project.

### RDD Classes

We will replace the RDD classes library with a library that bypasses the VODB() functions and will talk directly to the RDD layer, to gain a little bit of speed. Of course properties and parameters will also be strongly typed when possible.  
We will also add properties that allow to use an Enumerator on a workarea and add PropertyDescriptors and other information that a DotNet dataprovider needs to have.  
We may want to create 2 different DbServer classes, one with notification support and one without.

### Console, Internet, Report

These classes will also be updated. For example the Console classes will use the System.Console directly in stead of using the Win32 API Console API