Apollo

Core specification

Confidential

Revision number: 1

Date:

Name: P. van der Velde

Contents

[Disclaimer 2](#_Toc208933621)

[Introduction 2](#_Toc208933622)

# Disclaimer

This specification is not, by any stretch of the imagination, complete. It will need to be revised several times before it is complete. Currently several major parts are either missing or incomplete. This disclaimer will be updated to reflect any change in these sections. Finally a specification is supposed to be a ‘living’ document and therefore never complete. What were you thinking, better learn to live with this fact.

# Introduction

Apollo consists of 4 parts, the core system, the User Interface (UI), the project system and a series of plug-ins. This specification describes the core system only. For the specification of the other parts please read the appropriate specification documents. The goal of the core system is to provide services to the User Interface (UI) and the Project Sub-system.

Services provided are

* Plug-in & component tracking
* Creation of new projects, loading of existing projects
* Creation of project elements (data / generators / visualisers)

Furthermore some default services are provided like

* Messaging
* Exception handling
* Error reporting (call-home functions)
* Auto-update
* Commands
* Logging
* Start-up and shutdown
* Security
* License handling
* Provision of default utilities like class loaders for injection into different AppDomains
* Global application data (settings etc?)

Note that all interfacing between the core and other sections must be done through default interfaces and remotable information (no objects other than the .NET framework default ones)

In order to keep the core flexible and relatively easy to maintain a ‘non-pluggable’ plug-in architecture will be used. This means that the core will only load default additions, i.e. additions that are known at compile time. Obviously the core is a trusted part of the system and should thus only load our own stuff. Also extension of the core is (currently) not important.

All capabilities of the core are encapsulated in a ‘service’

* Describe start up
* Describe shut-down
* Describe messaging
* Describe logging
  + Needs to allow the user to read the log without knowing where the log is stored. So we’ll need a ShowLog command.
* Describe exception handling
* Describe security & license system
* Describe default utilities
* Describe data storage (plug-ins etc. All truely ‘global’ data)(could be in dbase)
* Describe plug-in tracking
* Describe creation of projects + linking to UI
* Describe creation of project elements

Note that using services for everything means that we have to specify relations and dependencies. Some services will have to be loaded before others. Also we have to do circular reference checking etc.

How will we connect services? Commands seem like they might be useful here. However how will we invoke these commands? They will have to be remotable and the results will have to be too. This also means that commands cannot be a service, but we might be able to have a service to register commands?

All commands need to be foolproof because their owners may die at any moment. Commands should be able to deal with this and still give a good response (probably something like ‘Owner died, service unavailable’).

Note that there are two levels of services. The base level services like from the second set and the more complex systems from the first set.

Note that if we use a database for storage of the global data then we might have a connection issue if we open the application twice (e.g. two CAD + plugin instances, or 1 CAD + plugin and 1 viewer, or … etc.). That is because even the in-memory databases use a file for storage. If we can coordinate the access (i.e. the database engine allows multiple connections to the file) then we can have a cool way of storing settings etc. because settings are always persisted.

We don’t want to have a single instance core shared with multiple apps because that could lead to having no core if the core start-up gets blocked somehow (either through a bug or though a DOS attack).

Also the GUI should use its own storage for storing settings etc. Don’t store those in the global data. Actually only the core should be able to write to the global data. Others may read but not write, or only write very specific data (e.g. plug-in meta data etc.) Probably not even. Plug-in specific data should be written in the project, not the core.

The core should only store predefined data (for plug-ins etc.). Never data that was generated during a simulation run.