Apollo

Overview specification

Confidential

Revision number: 1

Date:

Name: P. van der Velde

# 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

This specification will define the global architecture of the Apollo application. The Apollo application will consist of three different parts, the core, the user interface and one or more plug-ins. The core parts provide the administration of the plug-ins and the user interface components. Furthermore these parts also provide the communication methods for the different sections of the application. The plug-ins will contain the data generation, storage and visualization components.

# Architecture

The architecture of the Apollo application describes the individual parts, the core, the user interface and the plug-ins, and the manner in which these parts cooperate. The first part that will be described is the architecture of the core system, followed by the architectures of the user interface and the global architecture of the plug-ins.

## Core

The core of the Apollo application consists of a service provider and a set of services which provide all the capabilities for the system. The use of services allows adding and changing services without having to make changes to other parts of the application. It should be noted that the use of services by itself does not enforce loose coupling however it should make it simpler to have a loosely coupled system.

The main services are:

* **Core service:** This is used to allow the core elements (which are not services) to interact with the services on the service level.
* **Log service:** This is used to log information to one or more event logs. Services can request the creation of specific logs or just log to the general application wide logs.
* **Message service:** This is used to send information and requests between the services. Messages can be directed (from one service to another) or broadcast (from one service to whoever wants to know).
* **Timeline service:** This is used as the main collection source for all undo/redo capabilities. The timeline service tracks the order in which events happen and allows other services to track backwards or forwards along the timeline. Note that the timeline service does not store any of the data that is necessary to roll-back time.
* **Plug-in service:** This is used to track the available components.
* **Project service:** This holds all the project related capabilities.
* **User interface service:** This is used as the main entry point for the user interface. Holds most of the interface data structures and algorithms thereby allowing the real user interface to consist of nothing more than views and viewmodels.

All services will be constructed so that they can be loaded in a separate AppDomain in order to ensure that services can be fully unloaded and restarted. Furthermore the use of separate AppDomains should allow crashes to be contained thus allowing the service to be taken down without affecting the application too much.

### Log service

### Message service

### Timeline service

### Plug-in service

### The project system

* Based on data flow programming where the data ‘flows’ through different boxes that manipulate it.
* Have different generators for different types of data generation. Initially specify the simulation and experimental data generators
* A single experiment consists of data + generator + post-processing
* A single project consists of a Directed (Acyclic) Graph of experiments. There are different ways of creating DAG nodes. Always have a single top-level node(?)
* Features and components are contained in the generator? How about data?
* Should data only specify how/which data is written to storage and how/which data is retrieved. Then the storage can be our custom implementation.
* Data should be easily retrievable (see data storage spec).

### Core UI

* Provides the basic interface to Apollo for User Interfaces. The Core UI provides all the necessary commands and data views that allow a UI to perform all the requested actions
* Provides a way for UI’s to work with Apollo without having to know the internal structure, or to have to store data about the project. UI’s should never have to keep their own data structures. Data should only be stored in the project and kernel sections.
* ?

## User interface

## Plug-ins

* What is expected from the plug-ins (see the plug-in specification doc for more specific details)?
* What can plug-ins do?
* What can plug-ins not do?
* What are the rules for plug-ins?
* What area’s can be extended? Suggestion:
  + Experiment sub-elements
  + Data processing and visualization
  + Data import and export
  + Additional experiment types

## Interaction

* The core provides general services that can be used by both the UI and the plug-ins. The UI will have access to more different services than the plug-ins will.
* The core provides several different external API’s that can be used
  + User interface API. This API provides a way for the UI to interact with the core parts of Apollo.
  + Generic plug-in API. This allows components to have their own plug-ins. Defines a series of standard rules that allow the core to locate and classify plug-ins.
  + Project plug-in API. Allows developers to write project plug-ins. Project plug-ins provide extra types of generators and visualizers(?). This API is based on the generic plug-in API.
  + Component plug-in API. Allows developers to write components. The plug-in API ensures that plug-ins can be found by the system.