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 Apollo.

Apollo consists of 3 parts

* core
* User interface (UI)
* At least one but probably more plug-ins

Apollo gets all its power through the plug-ins. The core or the UI are not capable of any data generation or processing. The plug-ins provide the data generation, storage and visualization capabilities. The core provides tools for the administration of the plug-ins and the UI provides the interface with the user.

Describe:

* Interaction between the different parts
* Core
* User interface
* Plug-in behavior

# Architecture

This describes the general architecture of the different parts

## 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.

## Core

The core contains the kernel (services, global command system, global data), the project system and the core UI system. All three parts provide API’s which are at least within the core generally available. To the outside only a part of the project system API and the full core UI API are available at run-time. On top of that both API’s are only available to specific external components.

The project API is only available to two different types of plug-ins. The project plug-ins use one specific part of the API and the feature & component plug-ins use another part. Each type of plug-in will only be able to use their specific part of the API.

The user interface API is only available to user interfaces and not to plug-ins.

All parts should allow delay loading so that they system starts really quickly and only load those parts that are actually being used.

Kernel

* Provides essential capabilities to the project and core UI sections
* Provides a command engine that allows all clients to execute actions based on a command pattern
* Service based. All essential ‘things’ run as service except for commands
* Never directly used outside core. Only project and core UI can access.

Project

* 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 + visualizers
* 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?

# Working with Apollo

## Creating a new project

## Loading an existing project

## CAD plug-in

### Starting with an empty document

1. UI gets started by either the user (project explorer) or by a third party app (when running as plug-in to that app)
2. UI creates a core instance
   1. Kernel loads command system
   2. Kernel loads essential services only (logging, messaging, exception handler, licensing, call-home)
   3. Project system provides commands but does not load any sub-systems
   4. Core UI provides commands but does not load any sub-systems

### Opening an existing document after empty start

### Opening an existing document after having a normal CAD document

### Opening an existing document after having an Apollo-enabled document

### Starting with an existing document

### Saving an Apollo-enabled document

### Closing an Apollo-enabled document

## Project explorer

### Starting with no project

### Starting with a project

### Opening an existing project after empty start

### Opening an existing project after having a project

### Saving a project

### Closing a project

### Loading a experiment type

### Loading a experiment plug-in

### Loading a feature

### Unloading a feature

### Replacing a feature

### Loading a component

### Unloading a component

### Replacing a component

### Using undo/redo

### Creating a snapshot

### Returning to a snapshot

### Copying a snapshot

### Creating a sibling experiment

### Creating a child experiment

### Automatically creating child experiment