The Brahma Project Challenge

Design Documentation

Michael Eaton, Victoria Zheng

## Module Decomposition Diagram & Dependency Diagram

# 

The design for the Brahma Project consists of three distinct packages:  The Plugin package, the Control package, and the Foundation package.  The Plugin package is self-explanatory -- it contains the abstract definition for plugins, which the Control layer interacts with.  The Control layer manages the current UI, and manages the lifecycle of Brahma plugins.  The Control layer interacts with the Foundation layer, which details the system’s interactions with files stored on the system, such as detecting and accessing available plugins, and reading configuration data from the available configuration file.  
  
Inside the Control layer there is the PluginCore and the Launcher.  The Plugin contains the high level logic which interacts the panels it is presented by the plugins, and translates foundational data for display on the UI.  The Launcher deals with the DependencyManager and WatchDir to properly present the UI with useable plugins. The Dependency manager is a singleton which is retrieved through the DependencyRetreiver.

The Foundational layer contains logic for detecting and accessing plugins – the WatchDir class, and for resolving any plugin dependencies prior to loading them for user use. It also contains logic for reading and writing to the program configuration, the Configuration module.

## Class Diagram



Methods:

Plugin:

getId()  
            setId()

Layout()

dependenciesResolved()

setDependenciesResolved()

getDependencies()

addDependency()  
DependencyManager:  
            addPluginToLoadedPlugins()

areDependenciesResolved()

clearDependencies()

recheckAllDependenciesAndGetNewlyResolved()

removePluginFromList()

DependencyRetreiver:

getManager()

WatchDir:

processEvents()

ConfigurationManager:

getInstance()

getAllPropertyNames()

getPropertyValue()

setProperty()

resetColor()

savePropertyFile()  
Launcher:  
 loadBundle()  
 unloadBundle()  
PluginCore:  
 addPlugin()  
 removePlugin()

ConfigurationAccesser:

getBackgroundColor();

setBackgroundColor();

getPluginPath();

setPluginPaht();

LayoutAccesser:

getRightPanel();

setRightPanel();

getBottomPanel();

setBottomPanel();

## Interaction Diagram

Update the list of plugin



Launch Plugin



## Work Allocation

|  |  |
| --- | --- |
| Michael Eaton | Congnan Zheng |
| Design documents.  DependencyManager, DependencyRetreviever, PluginConstants, Plugin, dependencyTests, testPlugin, Launcher, changes to WatchDir, some changes to PluginCore. | Design documents.  ConfigurationManager,  config. properties, ConfigurationAccesser, LayoutAccesser,  Some changes to PluginCore. |

## Quality Attributes

We support the following attributes:

* Extensibility – Our design exposes the ability to create a threaded activity for usage in the central JPanel, as well as the ability to add to and modify the bottom and side JPanels. The configuration manager is exposed through the ConfigurationAccessor class to let the potential plugs access and modify the config.property files.
* Modifiability – Our design is strongly abstracts the base file-system and manages to implement a classic Model-View-Controller architecture. This allows a developer to easily grasp and modify portions of the code without seriously impacting unrelated portions.
* Usability – A plugin-based design allows the user to specify a number of configuration options through the UI, and expose an API for plugins to modify the behavior of the UI. This should allow the user to strongly influence their experience with the program and tailor it to their needs.
* Testability – The majority of our object-to-object interactions leverage standard java library objects, and produce easily testable outputs. This prevents the need for complicated testing schemes.