**Diagnostic Toolkit**

**Proof-of-Concept**

**ExxonMobil - Baton Rouge Chemical Plant**

**Document Version:**

**Oct 22, 2013**

Table of Contents

1. Change History 3

1.1 References 3

2. Introduction 4

2.1 Licensing 4

2.1.1 Ignition 4

2.1.2 JGraphX 4

2.2 Prerequisites 4

2.2.1 Java 4

2.2.2 Eclipse 4

2.2.3 Ignition 4

3. Architecture 5

3.1 Model Definition 5

3.2 Gateway 6

3.3 Designer 6

3.4 Client 6

4. Gateway 7

4.1 Gateway Functions 7

4.1.1 Dispatcher 7

4.1.2 Resource Changes 7

4.1.3 Block Execution 7

4.1.4 Tag Changes 7

4.2 Scripting Interface 7

4.3 TO-DO 8

5. Designer 9

5.1 NavTree 9

5.1.1 Root Node 9

5.1.2 Application Nodes 9

5.1.3 Diagram Nodes 9

5.2 Menu 10

5.3 Palette 10

5.4 Diagram 11

5.5 Blocks 11

5.6 Properties 11

5.7 Connections 12

5.8 Python 12

5.9 TO-DO 13

6. Client 15

6.1 Operator 15

6.2 Engineer 15

6.3 Communication with the Gateway 15

6.4 TO-DO 15

# Change History

| Version | Date | Author | Changes |
| --- | --- | --- | --- |
| 1.0 | Oct 22, 2013 | C. Coughlin (ILS) | Initial version |

## References

| Version | Date | Title | Author |
| --- | --- | --- | --- |
| 2.0r1 | May 2006 | ExxonMobil – BRCP – Toolkit Migration Project | ILS Automation for EM - BRCP |
| 2.0r1 | March 2007 | ExxonMobil – BRCP – Disgnostic Toolkit, Engineer’s Documentation | ILS Automation for EM - BRCP |

# Introduction

The ExxonMobile Chemicals Diagnostic Toolkit is designed for the detection, management, annunciation, and response to events. The output of any of the constituent tools is a diagnosis, a recommended response to the problem. Details of the current application may be found in the references.

The current project is a proof-of-concept that demonstrates the feasibility of conversion of the current Diagnostic Toolkit from a G2™ platform to one based on Ignition™ from Inductive Automation. This demonstration project shows how core features of these existing applications might be implemented on a totally different architecture. One of the primary challenges for any replacement system is that the new system support the flexibility of the old.

## Licensing

### Ignition

The application requires a commercial Ignition license to be obtained by ExxonMobil from Induction Automation. ExxonMobile is responsible for its installation on application servers.

### JGraphX

The Diagnostics application is heavily dependent on the graphics package, JGraphx available for download at http://www.jgraph.com/jgraphdownload.html. It is published under a 3-clause BSD license, details at http://en.wikipedia.org/wiki/BSD\_licenses. This license allows free and unfettered use of the package. JGraphX will be distributed by ILS Automatilon as part of the application delivery..

## Prerequisites

### Java

Use of the JGraphX library requires Java JDK1.7.

### Eclipse

For development support, compilation of Java 1.7 code requires Eclipse Juno, Kepler or newer.

### Ignition

This application requies Ignition 7.6.3 or newer. The IA-Labs-Scripting module, free from Inductive Automation is also required.

# Architecture

An Ignition project is, by its very nature, a client-server architecture. The server is called the “Gateway”. It supports autonomous processing without need for clients.

Client views are provided for onbservation and/or control of the application. Clients can either co-reside on the server platform or be remote.

The Ignition platform is customized by Python scripts – generally used to layout the user interface, and by add-on modules written in Java. The module code is generally used for the execution engine. The following diagram shows a functional breakdown between the Java and Python application layers. Java is shown in blue, Python in green

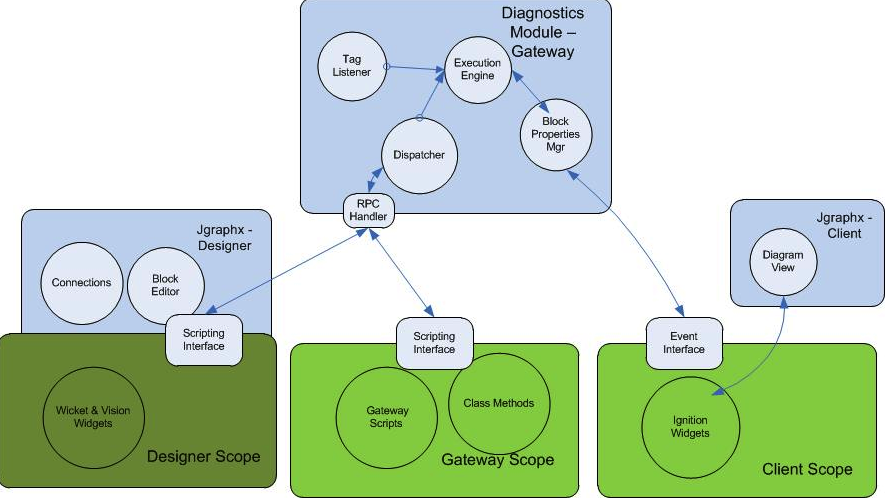


Figure – Collaboration Diagram

The module is a custom Java module based on the Ignition Software Development Kit (SDK). It executes completely within the Ignition gateway scope, but offers hooks via RPC calls to designer and client scopes as well.

## Model Definition

In a Model-View-Controller design, the model contains the definition of the core data structures, in this case, a block diagram. For the purposes of this application, the *model* is a JGraphX *mxIModel* object. It completely describes a diagram and can be used to render its diaplay. It becomes serialized into a project resource. This resource is the single unifying struction that binds knowledge of a diagram among all three Ignition scopes.

A diagram is identified by its TreePath value from the Designer’s NavTree. Within a diagram blocks (vertices) and connections (edges) have their own unique indices.

## Gateway

For the purposes of the diagnistic toolkit, the *Gateway* contains the engine that runs the logic blocks. Block logic is retained in separate Python class instances. These are created on diagram instantiation based on block property lists and Python class definitions in the scripting module.

This design allows the block logic to be saved and restored just as any other project resource. The gateway is a listener on the project resource changes. This is the mechanism for remaining in synch with the Designer.

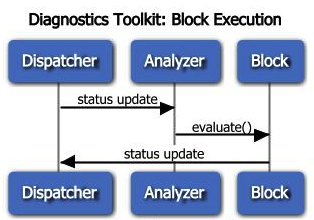


Figure – Sequence Diagram

The diagram above depicts the sequence of operation when a block status change is detected. The blocks are:

* Dispatcher – Scripting dispatcher. Receives notification from the Python layer of a block change.
* Analyzer – With knowledge of the diagram model, determines the next block(s) to be executed. Calls their evaluate method(s).
* Block – In the Python layer, a processing block. It’s *evaluate*() method, on completion, sends a status update message to the dispatcher.

## Designer

The *Designer* contains all code for creating and modifying diagrams.

## Client

Several *Client* views are provided for monitoring the state of logic blocks and other results.

## Serialization

Serialization refers to the process of converting Java objects into a format suitable for storage. The process of recovering the Java objects after storage is called “marshalling”.

Ignition provides an XML-based mechanism for serialization. However, this was not compatible with Java-generics (strongly typed lists) and was abandoned for that reason.

JSON is an alternative text-based solution. Tried GSON, a Google package. Nasty property of StackOverflow.

Settled on “Jackson”, also JSON. Apache license.

# Gateway

The *Gateway* “runs” the diagrams defined in the *Designer* scope. It is the keeper of the “model”, which is a description of the blocks in the diagrams, their attributes and states, and the connections between them.

While it may be tempting to think of the Gateway as a “running engine”, in fact, the Gateway code merely listens to asynchronous events and responds accordingly. This is shown in the sequence diagram,Figure 2.

## Gateway Functions

The subsections below describe the major controller classes in the Gateway scope.

### Dispatcher

The *GatewayRpcDispatcher* registers on startup as the receiver of RPC requests from client or designer components.

### Resource Changes

The *ModelResourceManager* is a project resource change listener. It detects updates to project resources that hold diagram model definitions. On resource change, it deserializes the model and informs the engine of the changes.

### Block Execution

The *BlockExecutionController* is a Singleton. It is the “engine”. Being a Singleton provides a well-known address for the object from anywhere in the Gateway. The engine is called when a block completes evaluation. Its function is to determine the block or blocks that are next to execute. The selected blocks are provided with the new output value, their inputt, thenthe *evaluate()* method is invoked.

### Tag Changes

The Gateway *TagHandler* subscribes to tags that are identified as block inputs. When the tags change, the handler informs the engine of an output change.

## Scripting Interface

The module supports a scripting interface for communication with Python code. The python-callable functions are:

system.ils.diagnostics.reportBlockCompletion

D e sc r i p t i o n

Inform the gateway when a block completes processing and has placed a result value on one of its output paths.

Syntax

reportBlockCompletion (path, index,value,output)

Parameters

String path – tree path to the diagram. The path is a colon-delimted string derived frin the Designer navigation tree.

int index – the index of the block within the diagram. This value in conjunction with the tree-path unquely identifies the block.

object value – the result of the block calculation that is to be propagated to blocks connected to its output.

String output – the name of the output connection upon which to propagate the result.

system.ils.diagnostics.enableDiagram

D e sc r i p t i o n

Enable or disable a diagnostics diagram. The initial state of a diagram is “enabled”.

Syntax

enableDiagram (path, flag)

Parameters

String path – tree path to the diagram.

boolean flag – true to enable the diagram

## TO-DO

ILS

* Write project change listener. Analyze project resource to extract diagram model.
* Write ModelAnalyzer that determines next block to execute given a status update.
* Write GatewayRPCHandler to receive RPC requests from the designer.
* Write Notifier to send block attribute changes to client scope.
* Call evaluation scripts on next block.
* Create tag listeners for entry blocks.
* Create python instances on a project resource update or addition.

IA

* Review overall design.

# Designer

The *Designer* is the only scope where changes to the model are supported.

## NavTree

The Designer’s navigation tree contains a “Diagnostics Toolkit” node. Use this tree to create new diagrams.

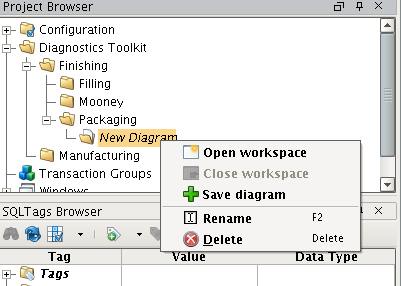


Figure – Navigation Tree

`The sections below summarize the available menu options for each of the three node levels.

### Root Node

* *Create New Application* – create a folder node that will contain a collection of application nodes.

### Application Nodes

* *Create New Diagram* – create a diagram node and workspace. A workspace is a container for blocks that will make an executable diagram.
* *Rename* – change the name of the family.
* *Delete* – remove the application node and all diagrams in it.

### Diagram Nodes

* double-click – opens the workspace.
* *Open* – open the diagram (also known as *Family*).
* *Close* – close the diagram. The user is prompted regarding a save.
* *Rename* – change the name of the diagram.
* *Delete* – remove the diagram node and associated workpace.

## Menu

The Designer main menu has an additional entry at the bottom of the “View” menu. This selection launches the palette containing blocks and connections to drag and drop into diagrams.

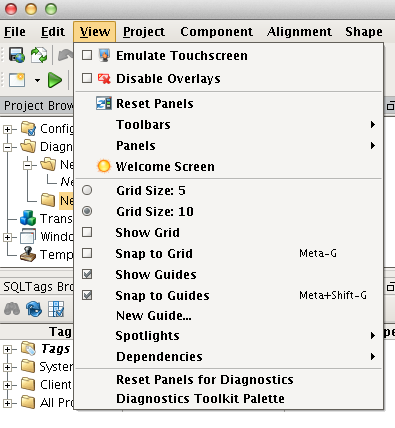


Figure – Designer Menu Addition

## Palette

The palette is an autonomous window launched by the menu selection described above.

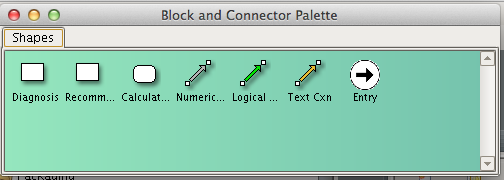


Figure – Palette

## Diagram

The diagram window is an autonomous window located within the confines of the Designer workspace area.

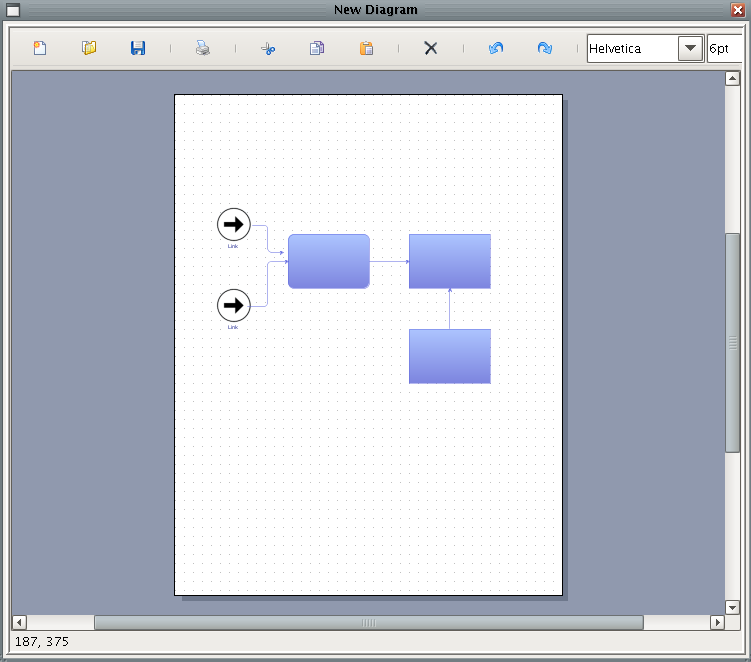


Figure – Diagram

## Blocks

<http://dia-installer.de/howto/create_shape/index.html.en>

## Properties

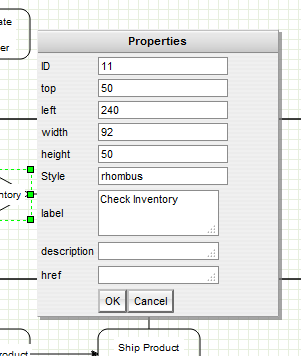


Figure – Properties Editor

## Connections

There are three types of connections:

* Numerical (data)
* Logical (state)
* Text (diagnoses, recommendations)

## Python

A python class is associated with each block type. The association of a python class instance with a block in the diagram is made at the time a diagram is saved. The python code provides the block properties as well as an evaluation method to be executed when the block receives values on its input ports.

# Copyright 2013 ILS Automation

#

# This class an abstract base class for all blocks in

# diagnostic diagrams.

#

class BasicBlock():

# Properties are a dictionaty of attributes keyed by name

properties = \

{'class':{'value':'app.diagnostics.classes.BasicBlock','readonly':True}}

# Input ports are named stubs for incoming connections

inports = []

# Outports are named stubs for outgoing connections

outports = []

# Return a list of property names.

def getPropertyNames():

return properties.getKeys()

# Return a specified property. The property

# is a dictionary guaranteed to have a "value".

def getPropertyName(name):

properties.get(name,{})

# Accept notification that a value has arrived on an input

# The default implementation does nothing

def setValue(value,port):

pass

# Evaluate the block. This default implementation

# does nothing.

def evaluate():

pass

Figure – Python Base Class

## Scripting Interface

The module supports a designer-scope scripting interface for communication with Python code. The python-callable functions are:

system.ils.diagnostics.getRepository

D e sc r i p t i o n

Return a dictionary for use in storing object instances that correspond to blocks in a diagram. The repository persists for the lifetime of the Designer.

The intent is for the repository to hold instances of Python classes keyed by: projectid:treePath:blockId

Syntax

getRepository ()

Return

PyDictionary repository – a common location to store instances of Python classes corresponding to blocks in a diagram.

## TO-DO

ILS

* Create python classes that correspond to blocks.
* Write registry for Python class instances to give them persistence.
* Create python class instances corresponding to creation of a diagram.
* Delete diagram, python instances and its project resource when the NavTree node is deleted.

JGX

* Write properties editor popup. Trigger on block right-click.
* Write custom renderer for blocks. Support different regions for animating.
* Support concept of stubs. An alternative might be a connection checker as a connection is attempted. Report to the user with a timed popup display(??) similar to an Android \_\_\_.
* Support connection to a particular stub. Consider a comparison block where the identy of the connection matters.
* Custom connector rendering.

IA

* Integrate editing of block attributes into standard Ignition properties editor. Attribute list must be changeable at run-time.
* Support “bindable” attributes.
* Trigger global save of all diagrams from designer save.
* How do I undo deletes of class instances, diagrams.

# Client

A *Client* view is the receiver of state or results for a diagram. The client comes in two flavors. The *Engineer* view is directed toward simulation and debugging. It features an animated “preview” view of a diagram that displays current status of the blocks. It cannot be edited.

The operator view emphasizes the display of conclusions and recommendations from the executing diagram. The operator’s windows feature standard Ignition components.

## Operator

The operator’s view ...

## Engineer

The engineer’s view

## Communication with the Gateway

Event listener ...

## TO-DO

ILS

* Modify Pete’s data pump as a tag-driver for test.
* Create tree widget that mimics the Designer NavTree.
* Write RPC handler to request/process the tree structure.
* Write PushNotificationListener to receive status change messages from the Gateway.
* Modify logging windows from Pete to log diagram status changes.

JGX

* Animate preview pane to show status of a diagram.

IA

* ??.

# Sample Application

blah, blah.

# Bugs

* When renaming a diagram node, a new resource is created. It should simply be that the old one gets a new name.
* Model resources are not part of the Export list.