CIMControlFramework

Developer's Guide

Date: January 19, 2011

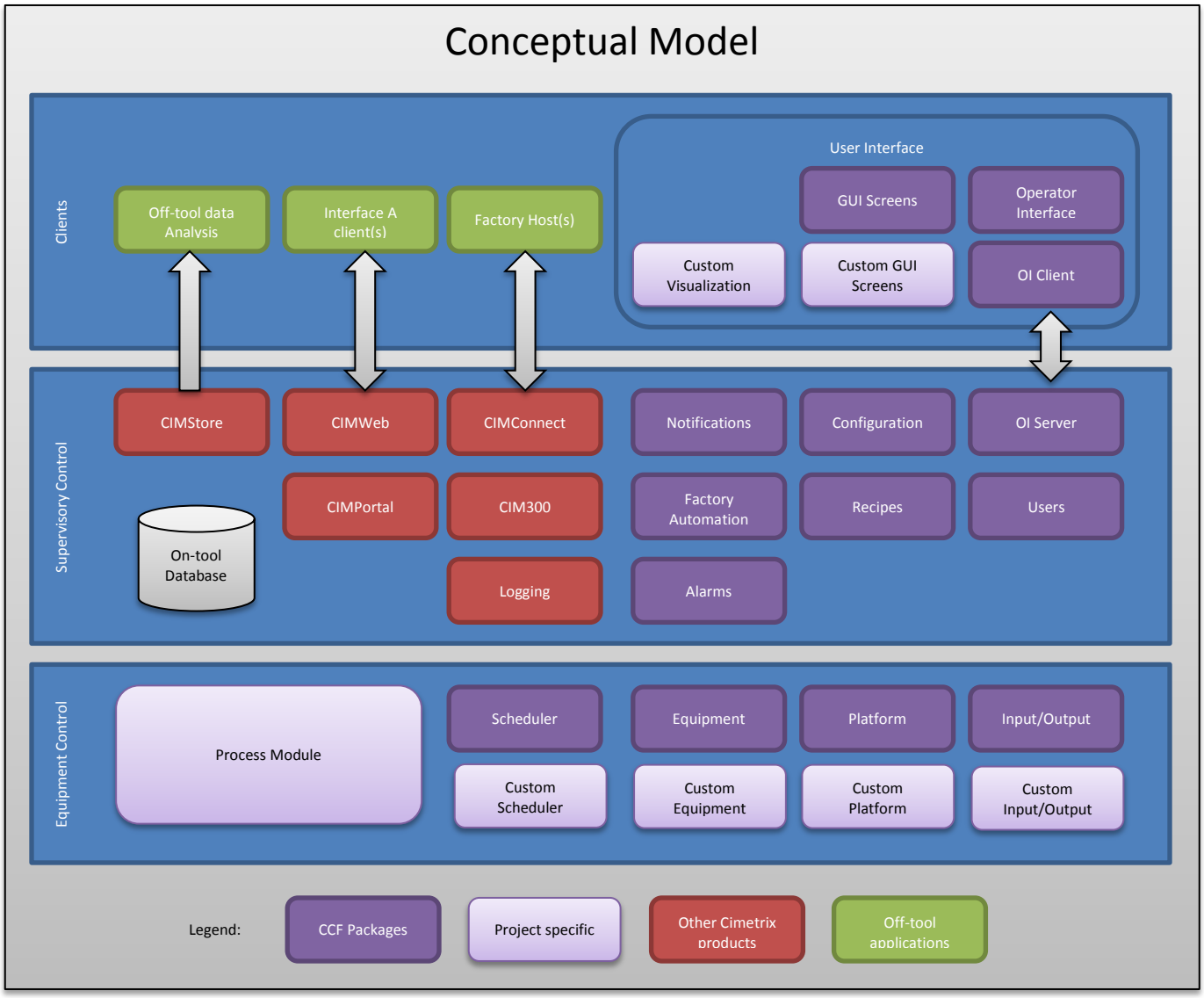
Table of Contents

1. Overview

CIMControlFramework (CCF) is a software development kit of software components, documentation, and processes that can be used to create an equipment control solution. As a framework, CCF allows the OEM or system integrators to include the CCF software modules which are appropriate for the solution, to remove or modify undesired behavior, and to support the solution independently of Cimetrix.

1.1 Architecture

The following diagram depicts the packages contained in CCF, separated into client, supervisory control, and equipment control functional groups.



1.1.1 Client Packages

1.1.1.1 Operator Interface (OI)

The Operator Interface application is a component of CIMControlFramework. The Operator Interface application provides the human-machine interface for the solution. The user interface is substantially compliant with SEMI E95. The Operator Interface application is a client to the services provided by the supervisory control software components. There may be multiple instances of the Operator Interface application (clients) providing independent interfaces to the supervisory control system. However, only one instance is permitted to control the equipment. Operator Interface application instances that are executed remotely (the computer is not on the same subnet as the computer running the supervisory control software) cannot control the equipment (view-only mode).

1.1.1.2 OI Client

The OIClient package contains functionality for connecting to the OIServer. It is hosted by the OperatorInterface application.

1.1.1.3 GUI Screens

The GuiScreens package contains user interface elements (e.g. information panel screens) which are common to all (or most) CIMControlFramework projects. These screens are hosted by the Operator Interface package.

1.1.1.4 Custom Screens

The CustomScreens package contains user interface elements (e.g. information panel screens) which are specific to a project. These screens are hosted by the Operator Interface package. This package allows customized screens to be developed for specific tools or projects.

Custom screens are created in Visual Studio. They are constructed visually so the finished product may be seen quickly, reducing the time required to create and customize GUI screens.

The CCF GUI framework provides functionality for attaching to system information and receiving updates. It also provides functionality for sending messages to execute methods on objects in any application in the system and receive return values. Screen access privileges are also a part of CCF.

1.1.1.5 Custom Visualization

The CustomVisualization package contains visualization controls which are specific to a project. These controls are hosted by the Operator Interface package, normally in custom screens. This package allows customized visualization controls to be developed for specific tools or projects.

1.1.2 Supervisory Control Packages

1.1.2.1 CIMPortal

CIMPortal is a Cimetrix product which provides data collection and distribution functionality.

1.1.2.2 CIMWeb

CIMWeb is a Cimetrix product. It is an optional CIMPortal feature which provides an equipment data acquisition (EDA) system and interface for equipment manufacturers. The CIMPortal/CIMWeb combination is being used in fabs around the world to provide a standards-compliant access mechanism enabling client applications to collect the equipment data needed to improve productivity.

1.1.2.3 CIMStore

CIMStore is a Cimetrix product. It is an optional CIMPortal feature which collects selected equipment data into an on-tool database. Currently only Oracle and MySQL relational database management systems are supported.

1.1.2.4 CIMConnect

CIMConnect is a Cimetrix product which implements the SECS/GEM communications interface between factory host system(s) and manufacturing equipment. CIMConnect implements (or partly implements) SEMI standards E4, E5, E30, E37 and E37.1.

1.1.2.5 CIM300

CIM300 is a Cimetrix product composed of software modules that enable tool control solutions to achieve compliance with SEMI 300 mm software standards including E39, E40, E87, E90, E94, and E116.

1.1.2.6 Logging

The logging package is a Cimetrix product which routes log messages to one or more destinations (e.g. log file, database, console). It also implements a WCF service interface that allows clients to route log messages to one central log repository.

Standard CCF screens allow access to the logging messages stored in the logging database and log files.

A LogViewer application is also provided for viewing log files independently of the Operator Interface. Additional visualization windows can be written for the LogViewer using plug-ins.

1.1.2.7 Recipes

The Recipes package provides recipe storage and retrieval services. These services are not dependent on recipe file format. Recipes can be divided into multiple namespaces (directories). The standard namespaces are Production and Engineering. Additional namespaces can be created in the Engineering namespace, but not Production.

The Recipes package also defines a sequence recipe and process recipe that can be used.

Standard CCF screens include a recipe management screen, a host recipe management screen, and a sequence recipe editor.

1.1.2.8 OI Server

The OI Server package implements a WCF service interface that connects its clients (typically instances of the Operator Interface application) to the rest of the system.

1.1.2.9 Users

The Users package implements a WCF service interface that allows clients to query or define user accounts, user groups and assign access rights to each. Administrators add and remove users and assign them to groups. The service also allows clients to determine the access rights of the current user.

A standard CCF screen is provided for user management.

1.1.2.10 Configuration

The Configuration package implements a WCF service interface that allows clients to retrieve configuration information, modify configuration settings, and be notified when a configuration setting is modified by another client. The service provides configuration data storage in the form of a single, XML-formatted file that is encrypted to discourage untraceable configuration data modification.

Standard CCF screens are included for making configuration changes and configuration management.

1.1.2.11 Alarms

The Alarms package implements a WCF service interface that allows clients to set, clear and recover alarms. Alarms are dynamically created by the components which use them. Alarms are reported to both the factory Host and InterfaceA.

Standard CCF screens include an Active Alarms screen where the operator can see which alarms are active and select recovery actions. Alarm History screens are also included where previous alarms can be viewed in either text or Pareto charts.

1.1.2.12 Factory Automation

The Factory Automation package provides services for managing substrates, control jobs, process jobs, and carriers. This package is responsible for achieving compliance with 300mm factory automation standards and depends directly on CIMConnect and CIM300.

1.1.2.13 Notifications

The Notifications packages implements a publish/subscribe design pattern. Typically it is used to route commands from the GUI to the software components which implement them. Adding new commands does not require this package to be modified.

1.1.3 Equipment Control Packages

1.1.3.1 IO

The IO package allows the status of individual IO channels to be queried and controlled. IO channels are identified by name (tag). The IO package has support for IO channels, groups of IO channels, and IO providers which serve as a hardware abstraction layer. IO providers for new types of hardware can be developed. The IO layer is a replaceable package.

1.1.3.2 Equipment

The Equipment package is composed of equipment control components. Each equipment control component implements an interface to an equipment system or sub-system. The lowest level equipment control components interface directly to hardware. Higher level equipment control components represent abstractions of groups of components. An equipment module is the highest level equipment control component. The Equipment layer is a replaceable package.

1.1.3.3 Platform

A Platform package is a collection of equipment control components which can control an equipment platform. Equipment platform consists of the equipment control components for a tool except for the process modules. A Platform package would be created if it were to be shared among multiple tools with different process modules. For example, if a platform from a vendor was to be shared across multiple tool lines. Platform packages are replaceable.

1.1.3.4 Scheduler

The scheduler package contains a sequencer that makes decisions dealing with the allocation of resources to tasks based on resource status at the moment of decision-making, using a set of rules designed to optimize system performance under a given set of assumptions and constraints. There are some stock schedulers included with CCF, but schedulers are often specific to a single project. The scheduler is a replaceable package.

1.2 Deployment Options

Many, but not all, CCF packages are designed to work with each other even if they are being used in other applications or other computers. This design gives a great deal of flexibility to the equipment OEM with respect to the number of computers and applications to use in the equipment. This section gives some example deployments and some guidelines on how to decide on a deployment model.

General design principles suggest keeping the number of applications and computers as small as feasible. Usually there is only a single CCF application on a computer because communications between packages is more efficient within the same application. The startup order inside a single application is also much easier to control. Additional computers should be added if hardware resources such as CPU, memory, or monitors, are being exhausted. Additional computers can also be added for security purposes, such as isolating operator interface computers from control computers. Remember that each additional computer requires additional

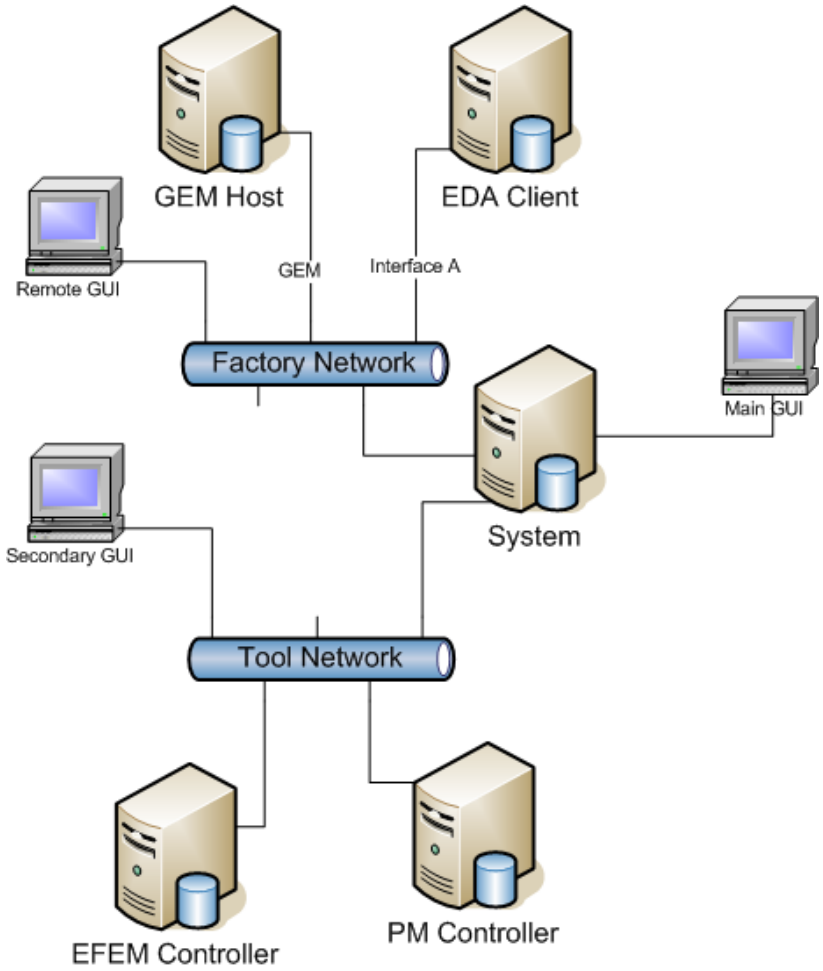
hardware, software licenses, configuration, startup management, and IT management. The non-hardware costs of additional computers are often underestimated.

1.2.1 One computer example

The simplest deployment option is a single CCF control computer. A single computer deployment could be appropriate when the equipment has:

- Moderate data publication rates or quantities
- Fewer process modules
- An ECS that requires typical interaction
- Moderate throughput

A deployment diagram for such a tool might look like this:



Note the following points:

- Even though there is a single CCF computer (System), there are other computers in the tool (EFEM Controller and PM Controller). The "Main GUI" computer may be the same computer as the System computer.
- The System computer is connected to both an internal tool network and a Factory network. It is common to have two non-bridged Ethernet controllers for this purpose. Dual Ethernet controllers eliminate any crosstalk between the two networks.
- Any additional CCF GUI computers would need to install some packages from CCF, but would not need a complete CCF installation.

1.2.2 Two computer example

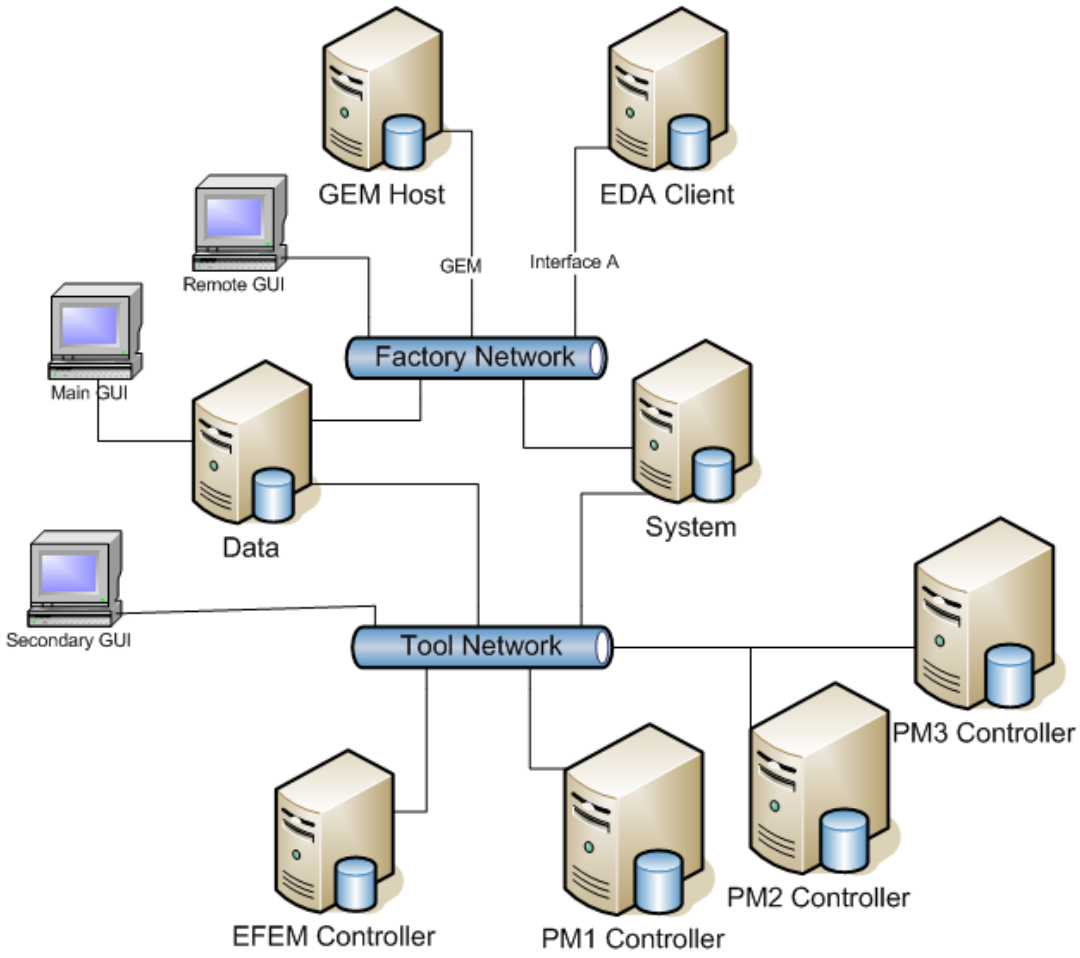
Another deployment option is two CCF control computers. A two computer deployment could be appropriate when the equipment has:

- A lot or faster data publication
- Many process modules
- An ECS that requires significant interaction or has stricter timing requirements
- A high throughput

The division of labor between the two computers might be:

1. System computer
    - Wafer handling
    - Factory Automation
    - Equipment Control
    - Scheduler
    - CIMConnect
2. Data computer
    - Main operator interface
    - Alarm package
    - Logging
    - Historical database
    - Configuration
    - Recipe
    - Users
    - CIMPortal
    - CIMStore

A deployment diagram for such a tool might look like this:

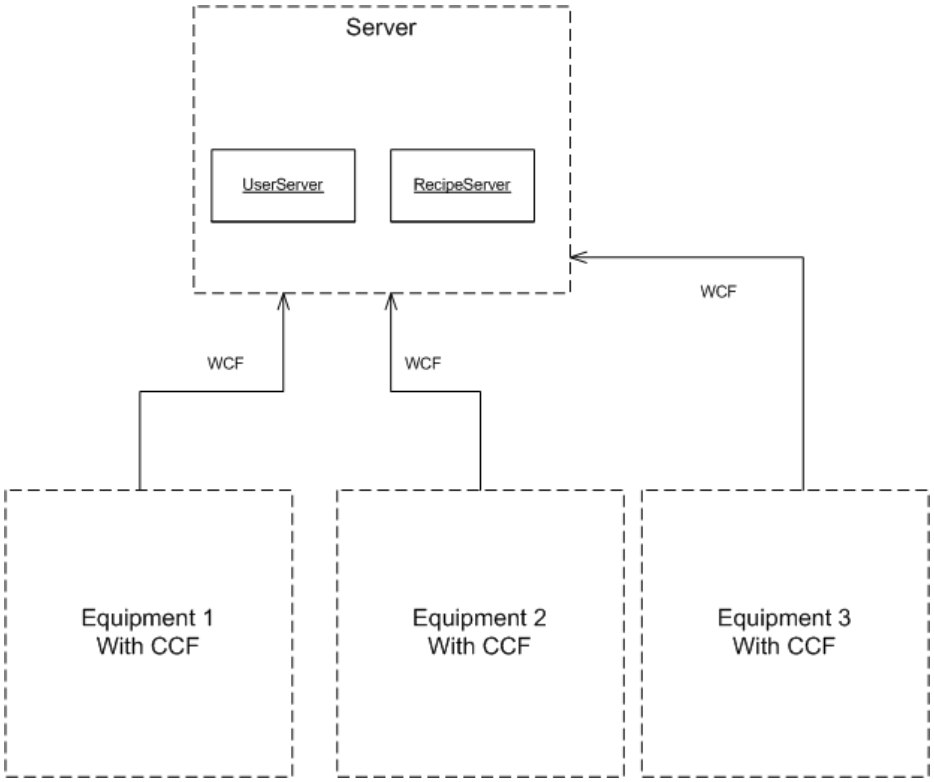


Note the following points:

- In this deployment, both the System and the Data computers are connected to both an internal tool network and a Factory network. This is required because CIMWeb, CIMStore, and CIMConnect all need to be visible to the Factory network.
- Any additional CCF GUI computers would need to install some packages from CCF, but would not need a complete CCF installation.

1.2.3 Multiple tool deployment options

In a multiple equipment deployment scenario, it is possible to run the UserServer and/or the RecipeServer on a computer that is accessed by more than one equipment. Any changes to the users and/or recipes would be reflected on all equipment. One possible diagram is shown below.



1.3 Source Code Folder Structure

NOTE: The base folder used to install CCF is referred to in this document as CCF\.

When CCF is installed, it creates several folders inside the base folder. These folders are explained in the following sections.

1.3.1 Dependencies

The Dependencies folder contains other software products, both from Cimetrix and 3rd parties, that are needed to use CCF. These dependencies are normally installed by the InstallationAssistant application during CCF installation.

The files in the Dependencies folder could be used to manually re-install the CCF dependencies on the developer's computer or to manually install them on a different computer.

1.3.2 UserDocuments

The UserDocuments folder contains the documents needed to use CCF. It also contains documents with information that can be placed in the Factory Automation manual and the Operators manual.

1.3.3 Source

The Source folder contains the CIMControlFramework source code and compiled assemblies. The assemblies are compiled using the Release | x86 platform settings and are in the CCF\Source\Shared folder. The CCF assemblies have all been created with a file and product version which indicates the build which created it.

The source code for most CCF packages is included in a folder named for the package in the CCF\Source folder. This folder contains a project for the package assembly and a test project. Examples of this layout are:

- Alarms
- CimStoreData
- Common
- Configuration
- Equipment
- History
- IdAssignment
- InstallationAssistant
- Notifications
- Recipes
- SampleSupervisor
- ToolSupervisor
- Users

A second set of folders in Source contain groups of related packages or assemblies, along with their test projects. Examples of this layout are:

- FactoryAutomation
- GUI
- IO
- Logging
- Platform
- Scheduler
- Simulation
- Utilities

The remaining folders have a variety of purposes. These folders include:

- CxCOMReferences—A project for generating COM Interop libraries for the other projects. Having a single project which generates COM Interop libraries and having the other projects reference the Interop libraries is more efficient than having each project generate its own COM Interop libraries.
- Equipment Models—A folder which contains files necessary for generating the CIMPortal package file, the EPJ file for CIMConnect, and the request definition files for CIMStore.
- Labs—A folder for projects related to CCF training.
- Shared—A folder which holds all the CCF assemblies. Also this is the root folder for running the CCF sample solution.

Some additional assemblies, such as the Cimetrix Logging and Value assemblies, are also included in the CCF\Source\Shared folder. The source code to these packages is not included in CCF. They are only included in the Source\Shared folder to increase ease-of-use.

The CCF\Source\Shared folder also contains subfolders which will be used by the default installation to store log files, recipes, and recipe archives. The locations of these folders may be easily changed in production systems using configuration parameters.

CCF can be compiled using either the CCF\Source\CIMControlFramework.sln (just the assemblies) or CCF\Source\CCF CI.sln (the assemblies and unit tests). These two solutions contain the projects for all

the CCF packages. Some packages also have a solution for just that particular package and its unit tests. These solutions are found inside the package folder.

There are several Visual Studio 2008 project and item templates available. They are in the CCF\Source\VS2008 Templates folder. To use them:

- Copy the contents of the "CCF\Source\VS2008 Templates\C# Project Templates" folder to your "My Documents\Visual Studio 2008\Templates\ProjectTemplates\Visual C#" folder. This makes the CCF project templates available to Visual Studio
- Copy the contents of the "CCF\Source\VS2008 Templates\C# Item Templates" folder to your "My Documents\Visual Studio 2008\Templates\ItemTemplates\Visual C#" folder. This makes the CCF item templates available to Visual Studio

1.4 Recommended Practices

- Each CCF release received from Cimetrix should be archived so you can reinstall that release if needed.
- Copy the CCF assemblies needed from the CCF\Source\Shared folder to a separate project source tree. Put them in the folder where the project assemblies will be built. In the project, reference the CCF assemblies from this folder and not where CCF is installed. These references should have the "Copy Local" property set to false. Unless you have a reason not to, you should set the "Use Specific Version" property to false as well.
- Do not make changes to CCF source if possible. Use extension and substitution with a different assembly if possible. If you must change the CCF source:
    - Copy the CCF source code outside the "Program Files" folder
    - Put the CCF source code in some type of revision control
    - Use a different product/file version than the standard CCF build number. Keep the first 3 numbers the same (Major version, minor version, service release), but do change the fourth number (build). You may want to start using much higher number. For example, if the standard product/file version is 3.0.0.154, your first custom build number might be 3.0.0.100001. The product/file version is stored in the Properties/AssemblyInfo.cs file.
    - After making changes, run the CCF unit tests as partial verification that nothing has been broken. Use the CCF\Source\CCF CI.sln. It contains all the active CCF unit tests.
    - Copy the new assemblies over to the project assembly folder.
    - Notify Cimetrix so the changes can be considered for inclusion in future CCF releases.
    - Consider implementing a method for tracking which CCF assemblies are from the install and which are custom built.
- Use your own namespaces in project assemblies. If your company does not have naming guidelines, you may wish to review the Microsoft naming guidelines. For CCF projects, namespaces starting with CompanyName. ProjectName would be appropriate.
- Do not use the prefix "Cimetrix" for project assembly names. Consider a prefix using your company name and/or your project name instead.

2. Programming

2.1 Configuration

There are several areas of configuration when using CIMControlFramework. They are discussed below.

2.1.1 CIMConnect

The CIMConnect EPJ file must be updated to include the parameters, events, and alarms produced by the equipment so the GEM Host can use them. This is not normally needed. Most parameters, events and alarms are added to CIMConnect while the CCF solution is starting.

2.1.2 CIMPortal

A CIMPortal equipment model must be created that represents the logical layout of the equipment and includes the parameters, events, and alarms produced by the equipment for Interface A Clients, CIMStore and the OperatorInterface. Most parameters, events, and alarms are added to CIMPortal while the CCF solution is starting.

2.1.2.1 CIMConnectDCIM

At a minimum, a CIMConnectDCIM instance must be created to add 300mm data and events from CIM300 to the equipment model.

2.1.2.2 CCFWCFAppDCIM

Equipment data is forwarded from FactoryAutomation to CIMPortal through the CCFWCFAppDCIM.

2.1.3 CIMStore

Any OEM desired data collection must be defined. For example, a trace for each process module that contains process data with ProcessingStarted, ProcessingComplete events is usually created. In addition, load locks and the VTM may have traces for pressure with PressureChangeStarted and PressureChangeCompleted events.

2.1.4 DatabaseLogSink database tables

A database table must be created for each configured DatabaseLogSink.

2.1.5 Configuration Parameters

Existing configuration parameters in the config.xml file may need their values adjusted. In addition new parameters for controls applications and new components may need to be added. You must set the System.EquipmentName parameter value to be the OEM equipment name used as the root of the CIMConnect EPJ file and the name of the CIMPortal equipment (CIMPortal deployment package name).

2.1.6 Users

Groups for the equipment will have to be defined in the groups.xml file. Users will need to be added to the users.xml file.

2.1.7 CCF Control Applications

Any WCF configurations for services and endpoints may need to be adjusted in the CCF Control Applications created for the integration project. The IP addresses and ports may need to be changed to match the deployment on the equipment.

2.2 New CCF Control Application

CCF Control Applications house CCF components. Cimetrix recommends console applications. These applications will create and configure the CCF servers such as logging and alarms as well as configure local

logging. The application configuration file (app.config) will need to contain WCF server and endpoint information for the servers and clients in the application. See the ToolSupervisor project as an example.

2.2.1 App.config

App.config contains application configuration settings. This is a file generated by Visual Studio when you add an Application Configuration File to your project using the Visual Studio wizard. If an application is hosting a CCF server the application's app.config file must contain the appropriate WCF server information. If an application is a client of a CCF server, the application's app.config file must contain a client endpoint for the WCF connection.

Server Example:

```
        <service name="Cimetrix.CimControlFramework.Recipes.RecipeServer">
          <endpoint address="net.tcp://localhost:6405" binding="netTcpBinding"
           bindingConfiguration="CcfDefaultBinding"
contract="Cimetrix.CimControlFramework.Recipes.IRecipeServer" />
        </service>
```

Client Example:

```
      <endpoint address="net.tcp://Data:6405" binding="netTcpBinding"
           bindingConfiguration="CcfDefaultBinding"
contract="Cimetrix.CimControlFramework.Recipes.IRecipeServer"
           name="RecipeEndpoint" />
```

2.2.2 Logging

Each application will have logging. The LogSinks for the application will need to be instantiated, configured, and added to Log. In addition, the app.config file may contain configuration for those sinks such as the LogCategory settings for each sink.

If the application is to send logging to a central LogServer, it should have a ClientLogSink instance and the app.config file should have an endpoint.

If the application is to host the LogServer, it should create a LogServer instance and the app.config file should contain server settings for the WCF connection.

2.2.3 Equipment Control

The application should instantiate any equipment control components it will contain. This includes platform and process control components (i.e. load port, load lock, robots, process modules, etc.)

2.3 GUI Customization

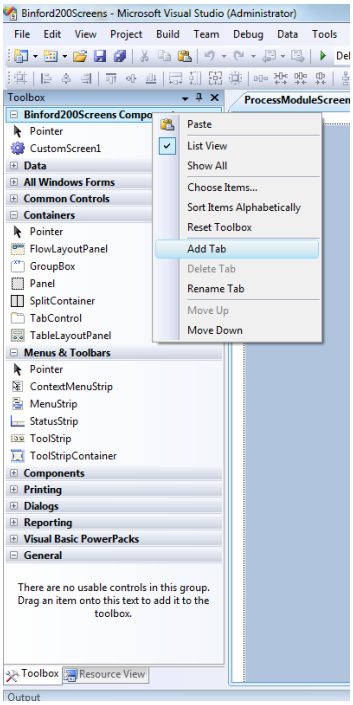
GUI Customization is done through screen DLLs loaded into OperatorInterface.exe. Typically, an integration project will involve creating custom screens for each process module type as well as recipe editor screens for each recipe type. Also, a custom system overview screen may be desired. A "CCF Custom Screen DLL" Visual Studio C# project template is available to start a new custom screen DLL. The GUI Customization lab uses this template.

2.3.1 CCF GUI Controls

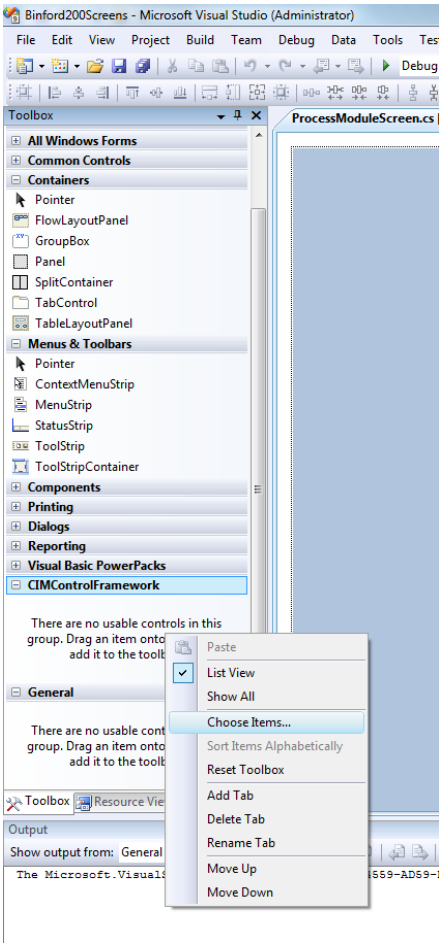
CCF has many GUI controls available for your screens. Given a choice, it is better to use the CCF GUI Controls as they are customized for CCF colorization, GUI privileges, and data binding. The best way to get started with them is to add them to the Visual Studio Toolbox so that you can drag them onto your screens when in the Designer view. Many of the controls are derived from standard Windows Forms controls and specialized for colorization, privileges, and data binding.

2.3.1.1 Add CCF Controls to the Toolbox

1. Right click on one of the tabs in the Toolbox and choose Add Tab. Make its name CIMControlFramework



2. Now right click in the empty space below the CIMControlFramework tab and choose Choose Items…



3. In the Choose Toolbox Items dialog, press the Browse button and navigate to the Shared folder and choose Cimetrix.GUICommon.dll and click OK. This will put all the CCF controls in the Toolbox under the CIMControlFramework tab.

2.3.1.2 CCF GUI Controls Overview

This section contains a brief overview of some of the CCF GUI control and how to use them.

Some of the common controls support localization and privileges. Those that support localization have a property named ResourceStringKey that is the string to look up in the resource file for the chosen language. This is set initially to the same value as the control text field. GUI privileges for input controls are done with the Privilege field. A privilege is just a string. If the logged in user has that string in his privileges collection, the control may be enabled, otherwise it will always be disabled. The privilege field may be left blank designating that no privilege applies. The control may still be enabled and disabled by screens based on equipment states, but disabling by lack of privilege will always take precedence.

2.3.1.2.1 BaseControl

BaseControl should not be used on any screen. It is a helper class used by visual controls and includes helper functions for localization, privileges, and gradient fills. When creating new custom controls, BaseControl makes a good base class if the new control is not already derived from an existing control like a TreeView.

2.3.1.2.2 BroadcastControl

BroadcastControl is a non-visual control used to route data changes to more than one control on a screen. It should not be put on a screen in the Designer; it should be created in code. CCF data binding on a screen relies on one parameter mapping to one control, it is a dictionary. In the case of multiple controls requiring the same parameter from the equipment, BroadcastControl is that one control used in data binding. The multiple controls on the screen are then registered with the BroadcastControl and will be notified by BroadcastControl when the parameter value changes.

2.3.1.2.3 CustomComboBox

CustomComboBox specializes System.Windows.Forms.ComboBox and adds privilege control and data binding. There is a property, PopulateMethod, to allow the screen to specify a function to be called when the databound value changed. This lets the screen determine how to process the value and populate the combo box items.

2.3.1.2.4 CustomGroupBox

CustomGroupBox is a custom group box control. It has a lot of visual options relating to title placement and look as well as the background fill color.

GroupBoxForeColor sets the group box title text color. GroupBoxBorderColor is the color of the outline box. GroupBoxBackgroundColor and GroupBoxBackgroundGradientColor are used to fill the background. GroupBoxHeaderColor and GroupBoxHeaderGradientColor are used to fill the title box background.

2.3.1.2.5 CustomListBox

CustomListBox specializes System.Windows.Forms.ListBox and adds colorization, privilege control, and data binding. There is a property, PopulateMethod, to allow the screen to specify a function to be called when the databound value changed. This lets the screen determine how to process the value and populate the list box.

The ListItemForeColor determines the color of the text in the list view and ListViewBackColor sets the color of the background.

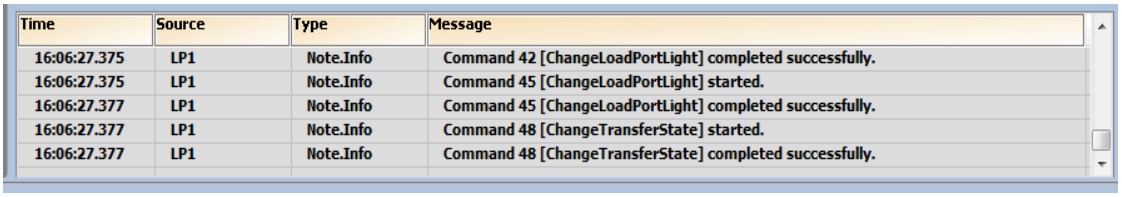
2.3.1.2.6 CustomListView

CustomListView specializes System.Windows.Forms.ListView and adds colorization, privilege control, and data binding. There is a property, PopulateMethod, to allow the screen to specify a function to be called when the databound value changed. This lets the screen determine how to process the value and populate the list view.

The ListItemForeColor determines the color of the text in the list box and ListViewBackColor sets the color of the background.

2.3.1.2.7 CustomListViewLogSink

CustomListViewLogSink is a specialization of CustomListView that is used to display live log messages in a Detail style list view. It may be configured to filter log message sources using the AddSources method or RegExString property. The RegExString property has precedence. Log messages may also be filtered by type using the AddType or AddTypes methods.



2.3.1.2.8 CustomRadioButton

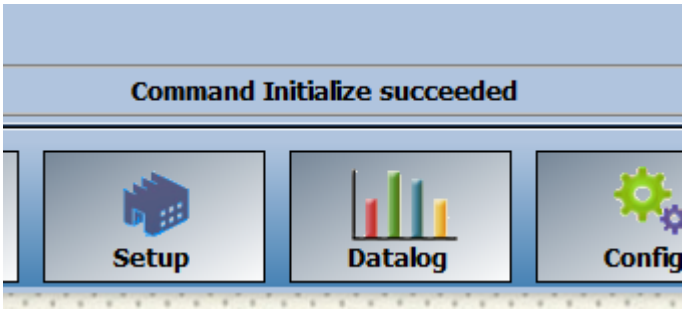
CustomRadioButton specializes System.Windows.Forms.RadioButton to add localization and privilege control.

ButtonForeColor sets the color of the label text.

2.3.1.2.9 CustomStatusBar

CustomStatusBar specializes System.Windows.Forms.StatusStrip to provide a status bar at the bottom of the screen to display GUI command activity for that screen. The sending of the command and the success or failure reply will be shown in the status bar. This can happen automatically by including the CustomStatusBar on the screen and setting the BaseScreen statusBar field to the instance of CustomStatusBar in the screen. BaseScreen will update the status bar as the screen calls ProcessCommand to send GUI commands to the equipment and when the response is received.

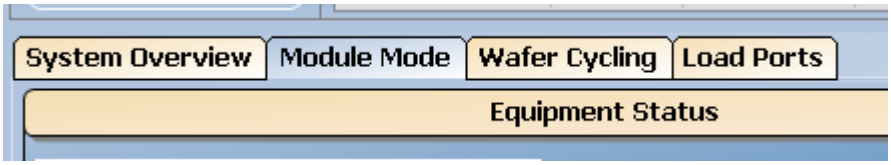
CustomStatusBar supports colorization. LabelDisplayForeColor is used for the text color and InformationPanelBackColor sets the background fill.



2.3.1.2.10 CustomTabControl

CustomTabControl adds colorization to System.Windows.Forms.TabControl. In addition, custom drawing has been provided for control of the colorization of the tabs.

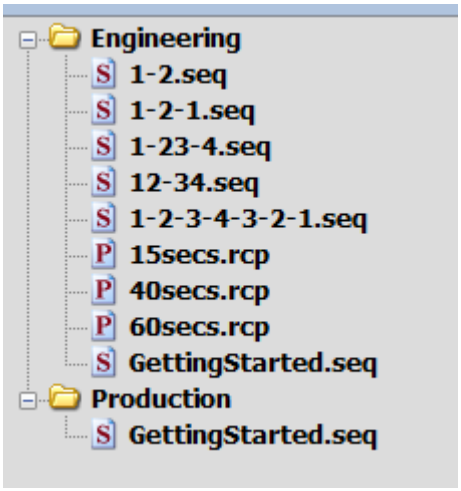
InformationPanelBackColor is used for the tab page fill color. TabBackColor is used to fill unselected tabs and SelectedTabBackColor is used for the selected tab. Tab text color is determined by the ForeColor property on each tab page and is not colorized automatically. The tabs backgrounds are filled with a gradient fill.



2.3.1.2.11 CustomTreeView

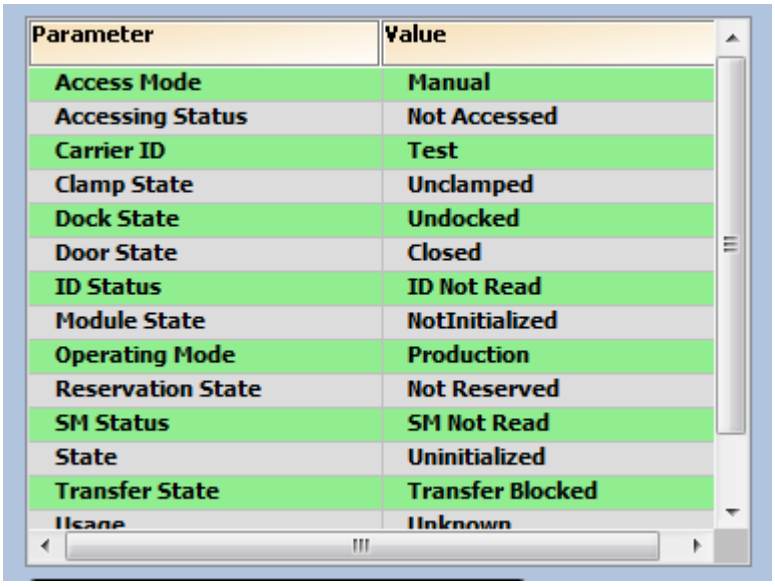
CustomTreeView specializes System.Windows.Form.Treeview to add privilege control, data binding, and colorization. There is a property, PopulateMethod, to allow the screen to specify a function to be called when the databound value changed. This lets the screen determine how to process the value and populate the tree view.

TreeViewBackColor is used for the fill and TreeViewForeColor is used for the text color.



2.3.1.2.12 DataListView

DataListView is a specialization of CustomListView that supports data binding of multiple parameters to the list view. The list view has two columns, one for the parameter name and the second for the parameter value. A screen uses one of the AddParameter methods to add parameters to the list view. When the last parameter has been added, the screen calls AddControlsCollection to set up the data binding.



Parameter	Value
Access Mode	Manual
Accessing Status	Not Accessed
Carrier ID	Test
Clamp State	Unclamped
Dock State	Undocked
Door State	Closed
ID Status	ID Not Read
Module State	NotInitialized
Operating Mode	Production
Reservation State	Not Reserved
SM Status	SM Not Read
State	Uninitialized
Transfer State	Transfer Blocked
Usage	Unknown

2.3.1.2.13 GateValveButton

GateValveButton is a specialization of ValveButton with an image more appropriate to a gate valve.

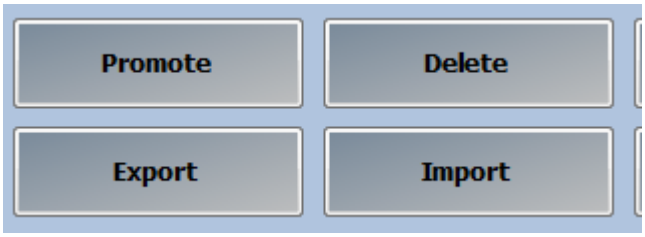
2.3.1.2.14 GaugeButton

GaugeButton is a specialization of GradientButton that include an image that changes color to reflect the reported state of the gauge as well as a pop-up menu with an Initialize command to allow the user to send the Initialize command to the device. The reported state is done by binding the button to a parameter that reports string values of: "Undefined", "Unknown", "Reading", or "Faulted". The image changes colors based on the state; Unknown and Undefined map to a grey image, Reading maps to a green image and Faulted a red image. The button has a CommandSelectedEvent the screen should handle to receive the pop-up menu choice from the button and route it to the device.

2.3.1.2.15 GradientButton

GradientButton is derived from System.Windows.Forms.Button. It has been specialized to have a gradient fill on the button. It supports having an image and text (both are optional) on the button and supports localization of that text. The button also supports a tool tip. GradientButton also supports IPrivilegedControl and has the Privilege property that may be set if not all users should have access to the button. The screen should use the normal button events to be notified if the user clicks the button.

ButtonForeColor sets the color of the button text and ButtonBackColor sets the button fill color.



2.3.1.2.16 HiddenControl

HiddenControl is a non-visual control used to route data changes to a method on a screen. It should not be put on a screen in the Designer; it should be created in code. This is useful if a screen needs to change non-CCF GUI controls based on the value, perform a calculation with the parameter value, or perhaps modify some aspect of the screen itself.

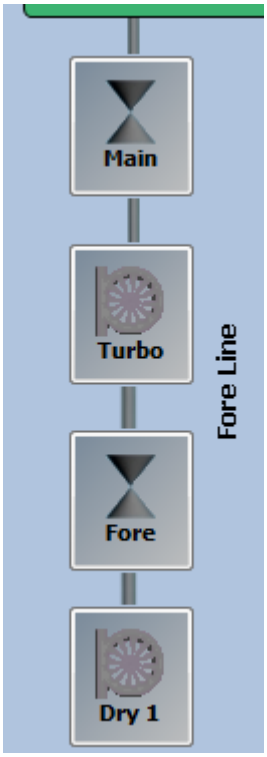
2.3.1.2.17 LED

The LED control mimics an LED. It shows a picture that changes color and an optional label next to it. The label may be on the right or left. The control supports data binding with values of 0 for off (grey), 1 for on (green), and 2 for alert (red). All numeric, binary, and Boolean parameters are supported. Ascii parameters are supported with the following mappings. The strings true, connected, normal, on, and open all map to on (green). The strings offline and faulted map to alert (red). All other string values will map to off (grey). The label supports localization.

LabelDisplayForeColor determines the color of the label text.

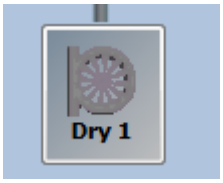
2.3.1.2.18 PipeControl

PipeControl is used to display horizontal or vertical pictures that represent pipes and are used in plumbing diagrams with ValveButtons, PumpButtons, and GaugeButtons. The control can display flow through color. It has a State property which may be set by the screen to change the color of the pipe. The control does not support data binding. The State changes the image as follows: Off is grey, On is green, and Error is red.



2.3.1.2.19 PumpButton

PumpButton is a specialization of GradientButton that include an image that changes color to reflect the reported state of the pump as well as a pop-up menu with Initialize, Open, and Close commands to allow the user to send commands to the device. The reported state is done by binding the button to a parameter that reports string values of: "Undefined", "Unknown", "Reading", "Opening", "Open", "Closing", "Closed", or "Faulted". The image changes colors based on the state; Closed is a grey image, Open is a green image, and Faulted a red image. The button has a CommandSelectedEvent the screen should handle to receive the pop-up menu choice from the button and route it to the device.



2.3.1.2.20 PushButton

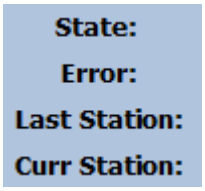
PushButton is a specialization of System.Windows.Forms.CheckBox. It is used for toggle buttons and displays a pushed state with a different color and visual appearance. Like

GradientButton, the background color is filled in a gradient. PushButton supports IPrivilegedControl. Localization of the button label is also supported. The button supports data binding to a Boolean value and will change state when the parameter changes. PushButton also supports images and tooltips.

2.3.1.2.21 TextDisplayField

TextDisplayField is a specialization of System.Windows.Forms.Label and is used for displaying text. It supports colorization. If the text is static, then localization is supported. It supports data binding and does its best to format the value using .NET ToString().

LabelDisplayForeColor determines the text color and LabelDisplayBackColor is the background fill.



2.3.1.2.22 TextEntryField

TextEntryField is a specialization of System.Windows.Forms.TextBox. It supports IPrivilegedControl to optionally prevent a user from entering text. It supports data binding and does its best to format the value using .NET ToString().

TextBoxForeColor sets the text color and TextBoxBackColor sets the fill color.

2.3.1.2.23 ValveButton

ValveButton is a specialization of GradientButton that include an image that changes color to reflect the reported state of the valve as well as a pop-up menu with a Initialize, On, and Off commands to allow the user to send commands to the device. The reported state is done by binding the button to a parameter that reports string values of: "Undefined", "Offline", "On", "Off", "Stopping", "Starting", or "Faulted". The image changes colors based on the state; Off is a grey image, On is a green image, and Faulted a red image. The button has a CommandSelectedEvent the screen should handle to receive the pop-up menu choice from the button and route it to the device.

2.3.1.2.24 VerticalLabel

VerticalLabel is a specialization of System.Windows.Forms.Label for text aligned vertically. It has a property to set either right facing or left facing text. It supports data binding and colorization.

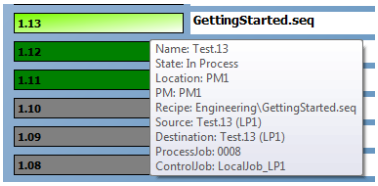
LabelDisplayForeColor sets the text color and LabelDisplayBackColor sets the background fill.

2.3.1.2.25 Wafer

Wafer is a specialization of System.Windows.Forms.Label and is used in conjunction with material map updates to display a wafer. A tool tip includes many of the properties of the wafer. The screen sets the Substrate property of the control with the Substrate from the material map update or null to update the control.

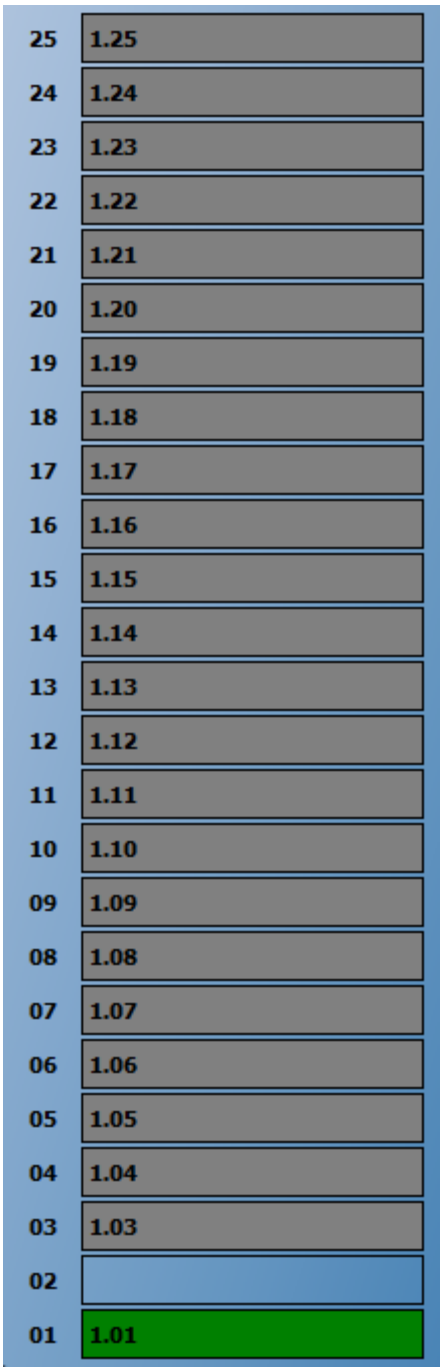
The control displays either the wafer ID or text that represents <loadport#>.<slot#> of the source load port and carrier slot from which the wafer was removed based on its DisplayIndicator property.

The control supports colorization. The background color of the control is used for state information. NoWaferStateColor is used if the slot is empty. If the wafer state is Aborted, Rejected, or Skipped then WaferMisProcessedColor is used. WaferInProcess is used for wafers being processed. WaferLostStateColor is used for wafers that have been lost. WaferUnProcessedStateColor is used for wafers that need processing. In addition, if the wafer is in a FOUP the background fill is solid, otherwise it is a gradient fill.



2.3.1.2.26 Wafers

Wafers is a custom control that displays a set of Wafers and slot numbers next to them. It is commonly used to display the contents of carriers. It is normally tied to material map updates. The screen calls the SetMap with the array of SubstrateLocationData for the carrier from the material map to update the control.



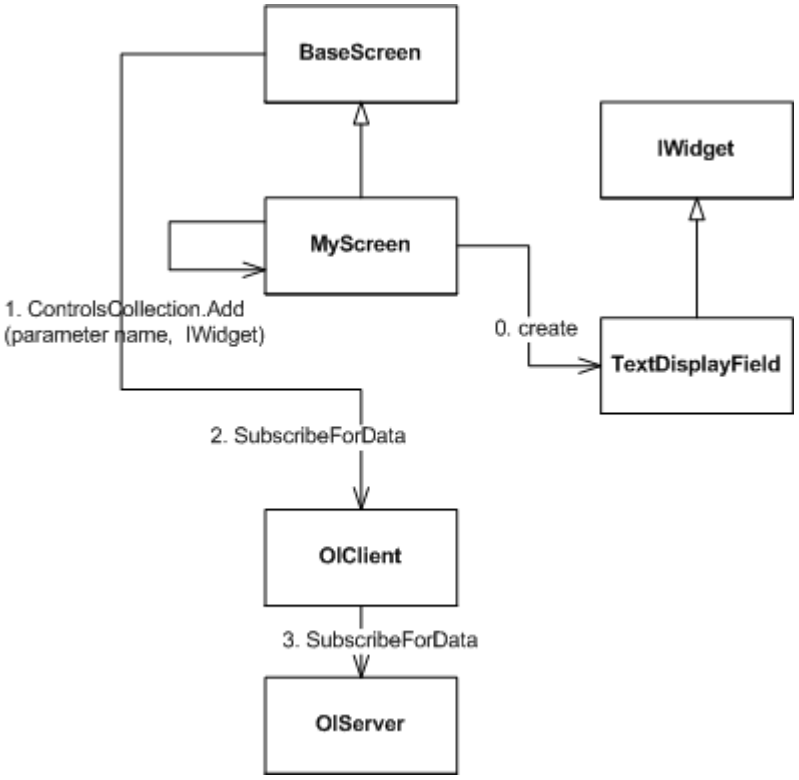
2.3.2 Add new screens

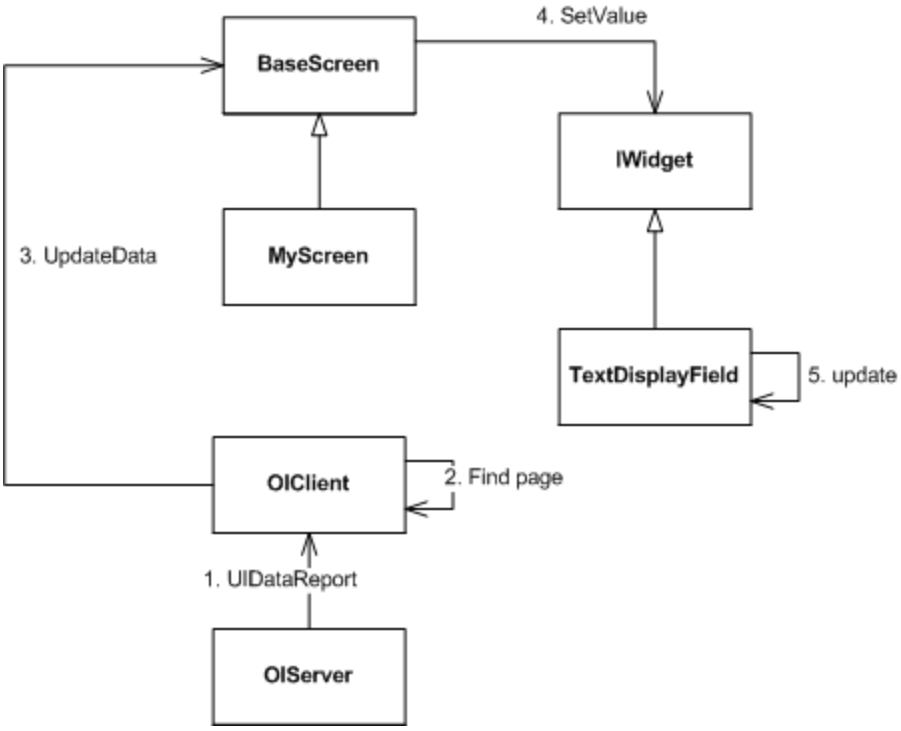
CCF Screens derive from BaseScreen. The easiest way to add new screens to your project is to use the "CCF Custom Screen template" from the Add New Item dialog in Visual Studio. The GUI Customization lab uses this template. The following is a summary of information about custom screens.

- Custom screens derive from BaseScreen instead of Form
- The screen constructor sets the BaseScreen screenName and codename for the new screen.

```
        public MyScreen(string name) : base(name, name)
```

- Screens usually use CCF controls (IWidget) added to the ControlsCollection property to bind them to data. Only one control may be added for a parameter. If multiple controls must be updated based on a parameter value, use a HiddenControl and route the data in the populate method.





- The GUI Customizer class in the custom screen DLL calls ICCFGuiLoader.AddScreenToScreenSet to add a screen to an existing screenset

```
MyScreen screen = new MyScreen("name");
gui.AddScreenToScreenSet("Jobs", screen);
```

2.3.3 Receive data to a screen method

This approach is used to receive a parameter to a function on the screen itself so that non-CCF controls may be updated based on the data.

1. Add an instance of HiddenControl to the screen for each parameter that must be routed to the screen. HiddenControl can be added to ControlsCollection. HiddenControl takes a delegate to a function within the screen that will receive the data.

```
private HiddenControl hcPressure;

public MyLoadLock() : base("MyLoadLock", "MyLoadLock")
{
    InitializeComponent();
    hcPressure = new HiddenControl();
    hcPressure.PopulateMethod = UpdatePressure;
    ControlsCollection.Add("Integra/LoadLock1/VacSys/CG1/LL1.VacSys.CG1.PressureInTorr"
, hcPressure);

}

private void UpdatePressure(IWidget control, CxValue value)
{
    double pressure = value.F8Value;

    lblPressure.Text = pressure.ToString();
```

```
    pbPressure.Value = (int)pressure;
    mypbPressure.Value = (int)pressure;

}
```

2.3.4 Custom GUI Control

The custom screen DLL can create custom controls that support CCF data binding following this procedure.

2. Add a new user control to your project
3. Modify the class to derive from IWidget and implement the interface. If it is only desired to add data binding to an existing control, derive from that control instead of UserControl.
4. In IWidget.SetValue, update the control as needed based on the value.
5. If you want the control to be disabled if the current logged in user is not privileged to see it, then implement IPrivilegedControl. The IPrivilegedControl.CheckPrivileges method will be called by the GUI framework whenever a user's privileges change so that the control may enable or disable itself appropriately.

2.3.5 GUI Commands

To call a GUI command from a screen, you must know the name of the command provider or target. You must also know the name of the command and what parameters are required. BaseScreen provides methods for working with GUI Commands. Call `ProcessCommand(string target, string name, Dictionary<string, CxValue> parameters)` to send a command. Override `ProcessSuccessGUICmdResponse` and/or `ProcessUnsuccessfulGUICmdResponse` to process GUI command replies.

2.3.6 Customize GUI Colors

You can change colors from your custom screen DLL by calling SetColors. The list of colors that are changeable are in the table below. Also, see the tutorial for more information on customizing GUI colors.

6. Modify GUI Customizer class ICCFGuiCustomizer.Load to add calls to ICCFGuiLoader. SetColors to change the desired colors

```
    Dictionary<string, System.Drawing.Color> colors = new Dictionary<string,
System.Drawing.Color>();
    colors["TitlePanelBackColor"] = System.Drawing.Color.BurlyWood;
    colors["TitlePanelLabelForeColor"] = System.Drawing.Color.Black;
    colors["NavigationBackColor"] = System.Drawing.Color.Coral;
    colors["InformationPanelBackColor"] = System.Drawing.Color.Gainsboro;
    colors["ButtonBackColor"] = System.Drawing.Color.MediumSeaGreen;
    gui.SetColors(colors);
```

Name	Description	Default Color
TextBoxBackColor	Backcolor for text boxes	Color [Gainsboro]
TextBoxForeColor	Text color for text boxes	Color [Black]
LabelDisplayForeColor	Text color for labels	Color [Black]
LabelDisplayBackColor	Backcolor of labels	Color [Transparent]
NavigationPanelBackColor	Backcolor of the navigation	Color [SteelBlue]

	panel	
TitlePanelBackColor	Backcolor of the Titlepanel	Color [SteelBlue]
InformationPanelBackColor	Backcolor of the information panel tabs	Color [LightSteelBlue]
TabBackColor	Tab background color	Color [Wheat]
SelectedTabBackColor	Selected tab backcolor	Color [LightSteelBlue]
ButtonBackColor	Button backcolor	Color [LightSlateGray]
ButtonGradientColor	Button background gradient fill color, gradient is from ButtonBackColor to ButtonGradientColor	Color [Silver]
ButtonForeColor	Button text color	Color [Black]
PushButtonPushedColor	Push button pushed color	Color [MediumSeaGreen]
PushButtonBackColor	Push button backcolor	Color [LightSlateGray]
PushButtonForeColor	Push button text color	Color [Black]
TitlePanelLabelForeColor	Text color to be used in title panel labels	Color [White]
TitlePanelLabelBackColor	Background color to be used in title panel labels	Color [Transparent]
CustomStatusBarBackColor	Backcolor for custom status bar	Color [LightSteelBlue]
CustomStatusBarForeColor	Text color for custom status bars	Color [Black]
AlarmSalienceColor	Color used for Alarm salience	Color [Red]
CautionSalienceColor	Color used for caution salience	Color [Yellow]
ProcessingSalienceColor	Color used for processing salience	Color [MediumBlue]
UserAttentionRequiredSalienceColor	Color used for user attention required	Color [MediumSeaGreen]
NoSalienceColor	Color used for no salience	Color [Transparent]
TreeViewBackColor	Backcolor for treeviews	Color [Gainsboro]
TreeViewForeColor	Text color for treeviews	Color [Black]
ListViewBackColor	Backcolor for listviews	Color [Gainsboro]
ListItemForeColor	Text color for listview items	Color [Black]
ListViewHeaderColor	Backcolor for listview headers	Color [Wheat]
ListViewBackColor2	Secondary backcolor for listview items for "green bar" look	Color [LightGreen]
ListViewSelectedColor	Selected listview item background color	Color [CornflowerBlue]
GroupBoxHeaderColor	Header color for group boxes	Color [Wheat]
GroupBoxHeaderGradientColor	Header gradient color for group boxes	Color [AntiqueWhite]
GroupBoxBorderColor	Border color for group boxes	Color [Black]
GroupBoxForeColor	Text color for group boxes	Color [Black]
GroupBoxBackgroundColor	Background color for group boxes	Color [LightSteelBlue]
GroupBoxBackgroundGradientColor	Background gradient color for group boxes	Color [SteelBlue]

TitlePanelGroupBoxBorderColor	Border color for group boxes in the Title panel	Color [White]
TitlePanelGroupBoxForeColor	Text color for group boxes in the Title panel	Color [White]
NoWaferStateColor	Color of the wafer when wafer slot is empty	Color [Empty]
WaferUnknownStateColor	Color of the wafer when state is unknowmn	Color [Magenta]
WaferUnProcessedStateColor	Color of the wafer when it is unprocessed	Color [Gray]
WaferLostStateColor	color of the wafer when it is lost	Color [Pink]
WaferMisProcessedColor	Color of the wafer when it is mis-processed	Color [Orange]
WaferSlotErrorColor	Color of wafer when it is mis-slotted	Color [Red]
WaferProcessedStateColor	Color of the wafer when it is successfully processed and back in carrier	Color [Green]
WaferInprocessColor	Color of the wafer when it is in process	Color [LawnGreen]
SystemOverviewLPDisplayBackColor	Backcolor of the system overview load port display	Color [White]
SystemOverviewLPDisplayForeColor	Forecolor of the system overview load port display	Color [Black]
SystemOverviewPMDisplayBackColor	Backcolor of the system overview process module display	Color [LightYellow]
SystemOverviewPMDisplayForeColor	Forecolor of the system overview process module display	Color [Black]
SystemOverviewLLDisplayBackColor	Backcolor of the system overview load lock display	Color [White]
SystemOverviewLLDisplayForeColor	Forecolor of the system overview load lock display	Color [Black]
LogErrorColor	Color for error log messages	Color [Red]
LogWarningColor	Color for warning log messages	Color [Blue]
LogDefaultColor	Default color for log messages	Color [Black]
SlotIsPickable	Color of pickable slots on EFEM Service and VTM Robot Service screens	Color [MediumSpringGreen]
SlotIsNotPlaceable	Color of unpickable slots on EFEM Service and VTM Robot Service screens	Color [Red]
OnColor	Color for items that are on or open	Color [Green]
OffColor	Color for items that are off or closed	Color [Gray]
FaultColor	Color for items that are faulted	Color [Red]

2.3.7 Add a New Screenset

A Screenset is a collection of screens presented in a tabbed control with one tab per screen containing the name of the screen. Screensets may be nested in screensets arbitrarily deep.

7. Modify GUI Customizer class ICCFGuiCustomizer.Load to add calls to ICCFGuiLoader. AddScreenSet

```
List<BaseScreen> screens = new List<BaseScreen>();
screens.Add(screen);
gui.AddScreenSet("MyScreens", screens,
SampleScreenDLL.Properties.Resources.MyScreenSetImage, screen.Name);
```

2.3.8 Modify Cimetrix.OperatorInterface.exe.config

You must add the new custom screen dll to the configuration file for the GUI. The ScreenDlls parameter contains a comma separated list of screen dlls to be loaded. More than one is allowed and they are loaded in the order specified in the string. Note that the last screen dll to change the GUI has priority. For example if three screen dlls are loaded and the first and third dll both change the NavigationPanelBackColor, the third dll's change will be used as it was last.

8. Modify the ScreenDlls application setting to include the new screen DLL. The value is a comma separated list of DLL names to load. No spaces around the commas.

2.4 GUI Visualizations

A visualization is an animated schematic view of one or more tool components that changes its appearance to reflect the tool's current state. For example, a visualization of a robot might show a simplified picture of the robot within its movement envelope. This robot visualization changes over time to show the robot's position in the tool, its orientation, and its payload. Some of the techniques used to communicate tool state in visualizations include:

- Changing image coloration
- Swapping images
- Changing image visibility
- Moving images (translation and/or rotation)

In the same sense that 'a picture is worth a thousand words', a visualization can often communicate the tool's state more effectively than a set of individual status indicators.

The CIMControlFramework's Operator Interface is implemented using Windows Forms technology, which is a proven and effective tool for creating Windows-based user interfaces. However, Windows Forms does not excel at tasks that are important in visualization, such as image manipulation and animation. For this reason, CIMControlFramework supports the creation of OI-based visualizations using Microsoft's newest GUI technology: Windows Presentation Foundation (WPF). WPF is based on DirectX technology and is vector-based, meaning that WPF can easily scale, translate, and rotate the elements of a visualization. It also includes built-in animation support and "pixel shader" capability, which allows for flexible and efficient manipulation of bitmaps.

The most effective Operator Interface screen is likely to be one that effectively mixes individual GUI controls (Windows Forms) with an occasional visualization (WPF). CIMControlFramework already has a large number of customized GUI controls (see Section 2.3.1) and there's no point in duplicating that functionality in WPF-based visualizations. Visualizations should be used to augment an Operator Interface screen in those cases in which an animated schematic view of the tool will be most effective.

Note: Creating GUI visualizations requires prior knowledge of WPF, and is not covered in this document. The descriptions that follow assume knowledge of WPF fundamentals.

2.4.1 Creating a new visualizations project

A visualizations project is a Visual Studio project that contains one or more visualizations. The easiest way to create a new visualizations project is to select the "CCF Custom Visualizations DLL" template in the New Project dialog in Visual Studio. The New Project wizard will create a new folder containing a solution including the new project.

Note: It's important that this solution folder be contained in the same parent folder as the CIMControlFramework's Shared folder. This is necessary because the project template will create relative references to DLL's in the Shared folder, and will also place the visualization's output DLL in the Shared folder.

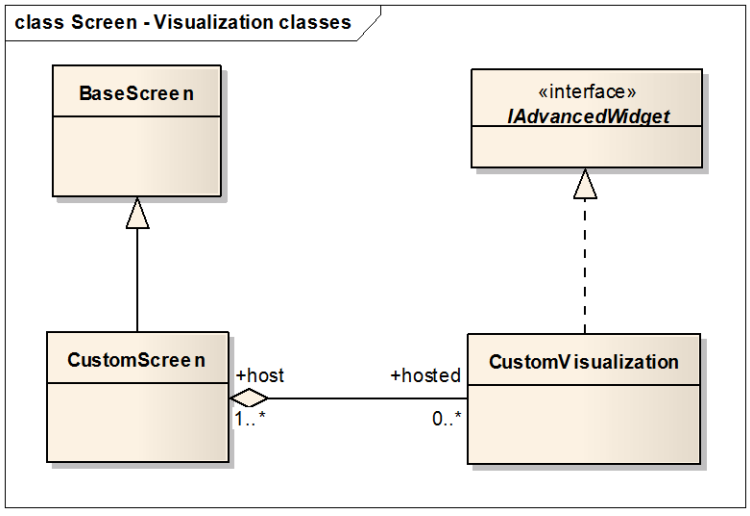
Once the project is created and opened in Visual Studio, you can add visualizations to it by selecting the "CCF Custom Visualization" template in the Add New Item dialog. Each added visualization will consist of a single XAML file and its associated 'code-behind' file. For example, if the new visualization is called "EfemFrontView", then the files "EfemFrontView.xaml" and "EfemFrontView.xaml.cs" will be added to the project. The "EfemFrontView.xaml" file defines a Canvas that will contain the graphical elements displayed by the visualization.

Note: By default, the new visualization will be a Canvas. This is because most visualizations are of fixed size and the graphical elements have a fixed location and don't require sophisticated layout capability. However, the use of a Canvas is not required, and any other subclass of System.Windows.Controls.Panel, such as a Grid or a DockPanel, can be used instead.

Every visualization in the project results in a public class. For example, the EfemFrontView visualization is compiled into an EfemFrontView class. An instance of that class can be instantiated and embedded into an Operator Interface screen, as described below.

2.4.2 Adding a visualization to an Operator Interface screen

An Operator Interface screen can host multiple visualizations as shown below.



A visualization is added to an existing Operator Interface screen in two steps:

1. Create the Windows.Forms.Panel that will contain the visualization display.
2. Create an instance of the visualization and connect it to the screen.

The following sections describe these steps in more detail. Although these steps describe adding a single visualization to an Operator Interface screen, any screen can host multiple visualizations.

2.4.2.1 Add a Windows.Forms.Panel to contain the visualization display

Open the solution which defines the Operator Interface screen and open the screen using the Windows Forms Designer. Select a Panel from the Toolbox and place it on the screen. Move the Panel to the desired screen location and size it to display the visualization. If the size of the Panel is different than the size of the visualization, the visualization will be scaled to fit the Panel.

2.4.2.2 Create an instance of the visualization and connect it to the screen

The following code example shows how a visualization is connected to a screen. This is typically done in the screen's constructor method.

```
// Create a TS400 visualization and connect it to this screen
TS400Visualization ts400Vis = new TS400Visualization(ScreenName);
this.AddControlToCollection(ts400Vis);
this.AttachPresentationElement(ts400Vis, ts400VisPanel);
```

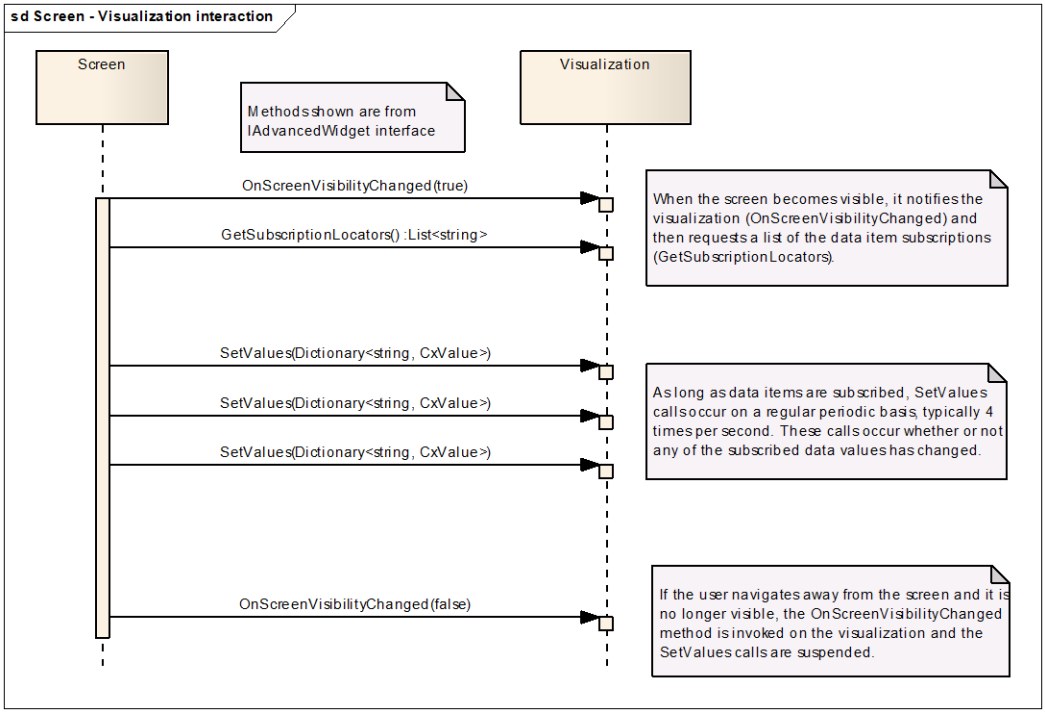
The first command creates an instance of the visualization class. This typically requires adding a project reference to the DLL that contains the visualization class. Adding a reference to a WPF-based visualization will also require that you have references to the following WPF assemblies:

- WindowsBase
- PresentationCore
- PresentationFramework

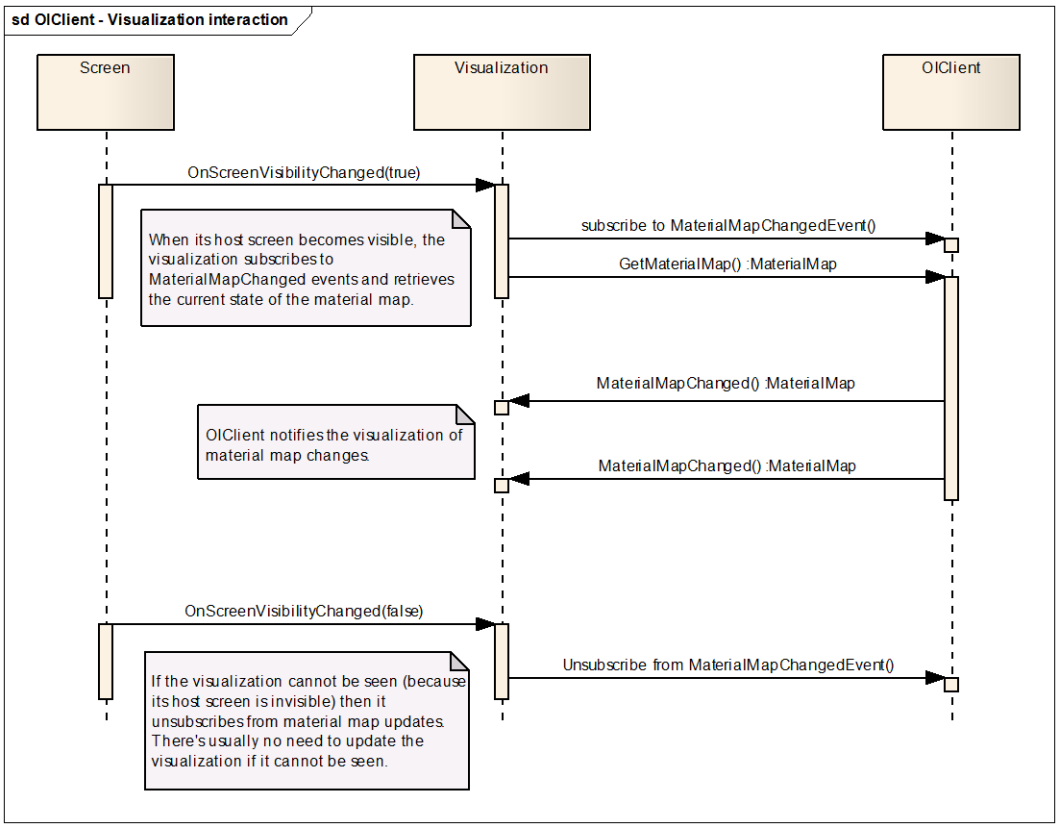
After the instance is created, it's added to a list of visualization controls kept by the screen. This is accomplished with an overload of the AddControlToCollection method. This method expects that the visualization implements the IAdvancedWidget interface, which is described in more detail in Section 2.4.3, below. The last command attaches the visualization to a Windows.Forms.Panel. This is the same panel added in Section 2.4.2.1, above.

2.4.3 Interactions between a screen and a visualization

The primary interaction mechanism between a host screen and a hosted visualization is through the IAdvancedWidget interface, which every visualization is expected to implement. This interface contains the methods that the screen uses to notify the visualization of screen visibility changes, to determine the visualization's data subscription needs, and to deliver regular periodic data value updates to the visualization. These interactions are illustrated in the following diagram.



In addition to the interactions shown above, a visualization may also subscribe to other types of data such as material map updates and job status changes. The following diagram illustrates how a visualization may interact directly with the OIClient instance (not through the screen) to acquire material map updates.



Finally, although the interactions shown above are the most common, a WPF-based visualization instance is a regular .NET object, so there are no restrictions on the interactions (property access, method call, events) that can occur between the host screen and the visualizations it contains. A visualization may define custom methods that are called by a custom screen, or raise events that are handled by a custom screen.

Important: Like Windows Forms, WPF only allows GUI elements to be manipulated on the GUI's thread of execution. Whenever a method is called on a visualization, it's important to transfer the call from the caller's thread to the WPF Dispatcher thread before manipulating the GUI's visual elements.

2.4.4 Debugging a visualization

Visualizations execute within an Operator Interface process and the Visual Studio debugger can be used to debug most visualization code, by following this procedure:

1. Start the Operator Interface and log in. At this point in time, all screens and their visualizations have been created.
2. Open the visualization solution in Visual Studio. Attach Visual Studio to the running Operator Interface by using the Debug => Attach To Process menu item.
3. Set a breakpoint in the visualization solution.

This works for most visualization debugging, however it isn't effective when debugging problems that occur before logging in, such as during initial screen loading and visualization creation. In this case, you must open the Operator Interface solution and start it in debug mode.

2.4.5 Scaling considerations

The visualizations added by the "CCF Custom Visualization" item template (see Section 2.4.1) create a single visualization class, and this class implements all of the visualization's display and behavior functionality. This simple structure is adequate for relatively simple visualizations, but is likely to become unwieldy and difficult to maintain for visualizations of medium to high complexity.

When creating larger scale visualizations, we suggest using one of the well-established user interface design patterns as a guideline for subdividing and organizing the visualization's functionality. The "Model – View – View Model" pattern (MVVM) is recommended. This pattern is intended for use with UI tools like WPF and is a specialization of the proven "Presentation Model" pattern.

2.5 Equipment Control System class

Some features require knowledge of and/or coordinated behavior between multiple software components of the equipment which don't have knowledge of each other. It often makes sense to collect these features into a single class or small set of classes. In CIMControlFramework, a class like this is often referred to as an "Equipment Control System" (ECS) class. Some functions that are commonly provided by an ECS class are detailed below. The term "Equipment Control System" is sometimes used to refer to the entire software component hierarchy used to control the equipment.

An example ECS class can be found at
Source\Platform\RorzeEquipment\RorzeEquipment\EquipmentControlSystem.cs.

2.5.1 Creation

The software hierarchy of components, subsystems, and modules that comprise the control system of the equipment usually needs to be created in a specific order. In addition, it is often the case that the exact type of component must also be known at this time. For example, it may not be enough to know that a component is a pump. The specific type of pump must be known. This knowledge is either contained by the ECS class or retrieved from configuration by the ECS class.

Creation of the software component hierarchy is typically done as part of the control application startup.

2.5.2 Initialize Service

Equipment initialization has order constraints. Normally, robots need to be initialized first to ensure the robot arms are not protruding into a gate valve or door. Shared components such as vacuum systems are usually initialized before the components which share them. Generally speaking, in the software hierarchy, the children are initialized before the parents.

Initialization is typically initiated by an operator using the operator interface.

2.5.3 Material Movement

The material movement screen on the operator interface issues several types of commands. These commands are normally processed by the ECS class.

2.5.3.1 MoveWafer Service

This service request is to move a wafer from one substrate location to another. The source substrate location and destination substrate location are not required to be adjacent. This service request frequently results in multiple pick and place operations.

2.5.3.2 ChangeMaterialMap Service

This service request is to change the material map. This command could result in a substrate being added to the material map, removed from the material map, or moved from one location to another in the material map. This is done strictly as a change to the material map. No physical motion is commanded.

2.5.4 Material Transfer

Material transfer commands are initiated by the operator using the operator interface. These commands are typically a single operation for a robot (Pick, Place, and Swap), so are typically for the robot to interact with an adjacent substrate location. This command is implemented by the ECS class instead of the robot because the robot must interact with a target module.

2.5.5 General Service Commands

Any general service command may be implemented by the ECS class.

2.5.5.1 ListModules

The operator interface requests the ListModules service to retrieve a list of modules in the software hierarchy. The ECS class is a good location for this command because it has knowledge of all the modules.

2.6 New Wafer Handling Platform

To support a new wafer handling platform, drivers need to be developed that will interact with the new platform. This section describes how to develop drivers for various platform components, such as load ports.

2.6.1 Load Port

A new type of load port is supported by developing a new load port driver, which consists of two parts:

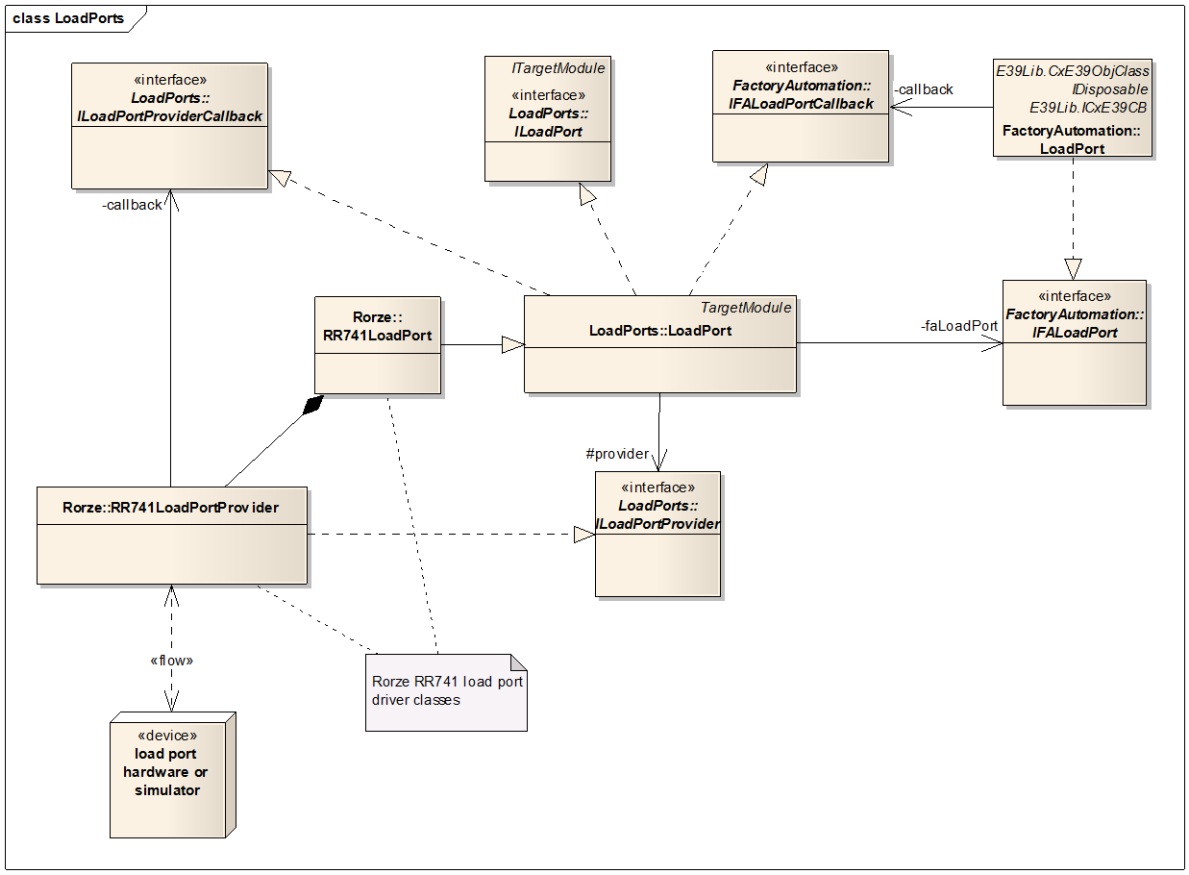
- A LoadPort class
- A LoadPortProvider class

For most drivers, the LoadPort class will inherit most or all of its required functionality from its CCF-provided superclass, and the bulk of the driver development work will be to create an appropriate LoadPortProvider class.

Note: In the following description, the terms LoadPort and LoadPortProvider are used to refer to the two classes that comprise the load port driver however those classes will have actual names that reflect the type of load port they support. For example, the driver for the Rorze RR741 load port consists of

RR741LoadPort and RR741LoadPortProvider classes. The driver's LoadPort class should not be confused with the CCF class named LoadPort in the Cimetrix.CimControlFramework.Equipment.Modules.LoadPorts namespace. When referring to that CCF class, the fully qualified name will be used to avoid confusion.

The following class diagram illustrates the relationships between the load port driver classes and other relevant CCF classes. Note that the LoadPort class, in this case RR741LoadPort, 'contains' the LoadPortProvider class, named RR741LoadPortProvider. In this diagram, the RR741LoadPort class invokes load port command methods, such as PerformClamp, on the ILoadPortProvider interface implemented by RR741LoadPortProvider. In turn, RR741LoadPortProvider informs RR741LoadPort of load port status changes by invoking status update methods defined in the ILoadPortProviderCallback interface. In this way, the driver's LoadPort class issues commands that the driver's LoadPortProvider communicates to the load port hardware, and the LoadPortProvider provides load port status updates back to its corresponding LoadPort class.



The following two sections describe the responsibilities of the driver's LoadPort and LoadPortProvider classes in more detail.

2.6.1.1 LoadPort responsibility

The driver's LoadPort class is responsible for overall load port behavior and for disseminating load port status data to other parts of CCF. This class should be a subclass of the Cimetrix.CimControlFramework.Equipment.Modules.LoadPorts.LoadPort class and typically, inherits all required functionality from that superclass. The LoadPort class is also responsible for instantiating and keeping a reference to an instance of the driver's LoadPortProvider class and for

providing that LoadPortProvider with an implementation of the ILoadPortProviderCallback interface (itself). Typical code for a LoadPort class constructor is shown below:

```
/// <summary>
/// Construct a Rorze RR741 load port
/// </summary>
/// <param name="name">name for the load port module</param>
/// <param name="falp">reference to the FactoryAutomation.LoadPort for this
port</param>
/// <param name="configPath">root path in the config file for configuration
parameters for this load port</param>
public RR741LoadPort(string name, FactoryAutomation.IFALoadPort falp, string
configPath)
    : base(name, falp)
{
    this.provider = new RR741LoadPortProvider(name, this, configPath);
}
```

The LoadPort constructor is typically invoked once for each of the tool's load ports in the CreateLoadPorts method of the EquipmentControlSystem class. One pair of LoadPort/LoadPortProvider instances is created for each physical load port present on the tool.

2.6.1.2 LoadPortProvider responsibility

The driver's LoadPortProvider class is responsible for communicating load port commands to the load port hardware. These commands are invoked on the LoadPortProvider via methods in the ILoadPortProvider interface.

The LoadPortProvider is also responsible for collecting up-to-date load port status from the load port hardware. This status may be collected by polling the hardware and/or by event messages sent by the hardware. When the LoadPortProvider receives a load port status update, it updates CCF with the latest status by invoking callback methods defined in the ILoadPortProviderCallback interface. An implementation of ILoadPortProviderCallback is provided to the LoadPortProvider as a constructor argument.

Note: All communication with the load port hardware is handled by the driver's LoadPortProvider class. This communication is the primary role of the LoadPortProvider.

Lastly, the LoadPortProvider class is responsible for defining and updating any special data variables, events, and alarms associated with the load port. The superclass of each LoadPort class, Cimetrix.CimControlFramework.Equipment.Modules.LoadPorts.LoadPort, already provides a standard set of data variables, events, and alarms for each load port, so most LoadPortProvider implementations will not need to do this. However, if additional data items are required, these are implemented in the driver's LoadPortProvider class.

The LoadPortProvider class is usually not responsible for enforcing load port sequencing behavior or constraints. For example, when told to 'open the FOUP door', the LoadPortProvider class simply issues the command(s) to open the door to the load port hardware. It does not need to check that the FOUP is currently docked before attempting to open the FOUP door because that sequencing behavior is the responsibility of the LoadPort class.

Exception: In special cases, the LoadPortProvider may need to check load port state or the state of an interlock before issuing a command. For example, it's obviously important that the load port door is not closed while a robot is accessing substrates in the FOUP at that load port. If sufficient

interlocks are not provided by the hardware or hardware controllers, special handling to prevent this situation may need to be added to the LoadPort or LoadPortProvider classes.

Detailed descriptions of the methods defined in the ILoadPortProvider and ILoadPortProviderCallback interfaces can be found in the source code comments attached to those methods in the ILoadPortProvider.cs file, in the CCF Equipment solution.

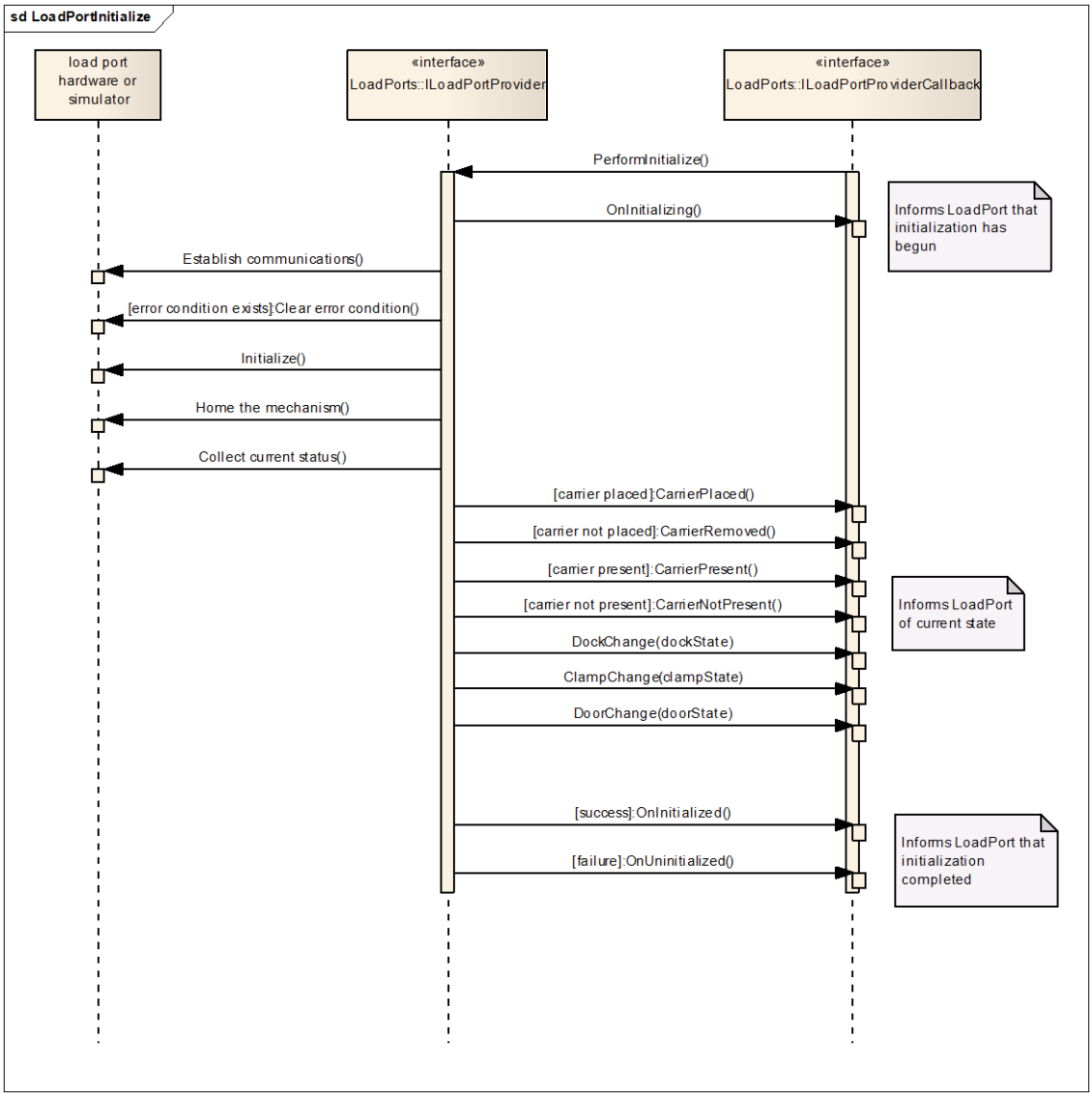
2.6.1.3 Initialization sequence

When the ILoadPortProvider.PerformInitialize method is called, the LoadPortProvider should do the following:

1. Call the ILoadPortProviderCallback.OnInitializing method to inform the LoadPort that initialization has begun.
2. Establish the communication connection to the load port controller.
3. Attempt to clear any pre-existing error condition that might exist. This is required because re-initializing is sometimes done to try to rectify an otherwise unresolvable error condition.
4. Perform any initialization necessary to insure that the load port is ready to be used and that load port status will be collected and kept current.
5. 'Home' the load port mechanism, if applicable.
6. Collect the current status of the load port (present/placed/docked/clamped/opened) and call the appropriate ILoadPortProviderCallback methods to update the LoadPort with this current status.
7. Call the ILoadPortProviderCallback.OnInitialized method to inform the LoadPort that initialization has succeeded.

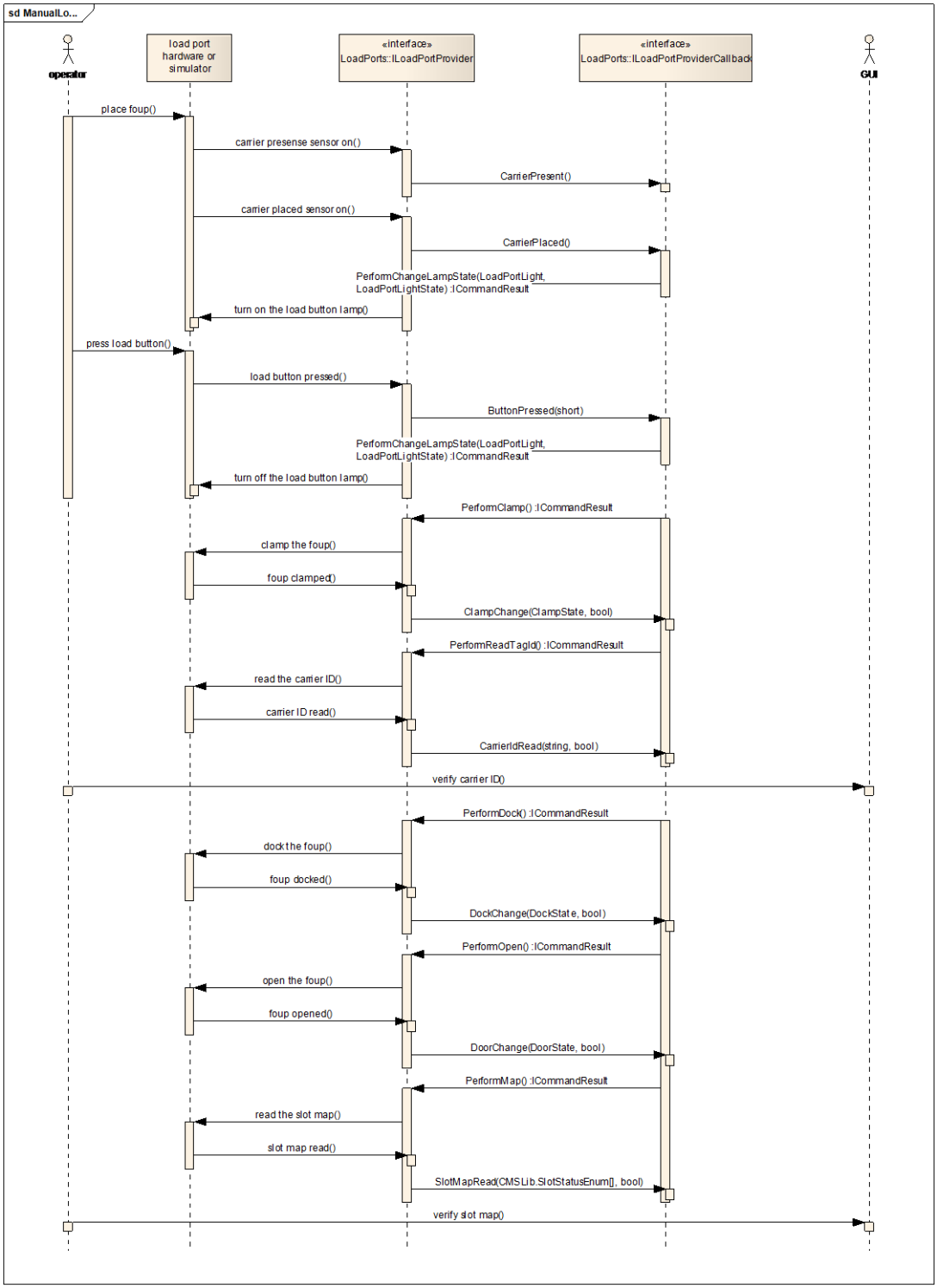
If initialization fails, the ILoadPortProviderCallback.OnUninitialized method should be called instead of OnInitialized.

The following sequence diagram illustrates the interactions between the load port hardware and the LoadPort and LoadPortProvider classes when the load port is initialized. In this diagram, the LoadPortProvider class is identified by the interface it implements: ILoadPortProvider, and the LoadPort class is identified by the interface it implements: ILoadPortProviderCallback.



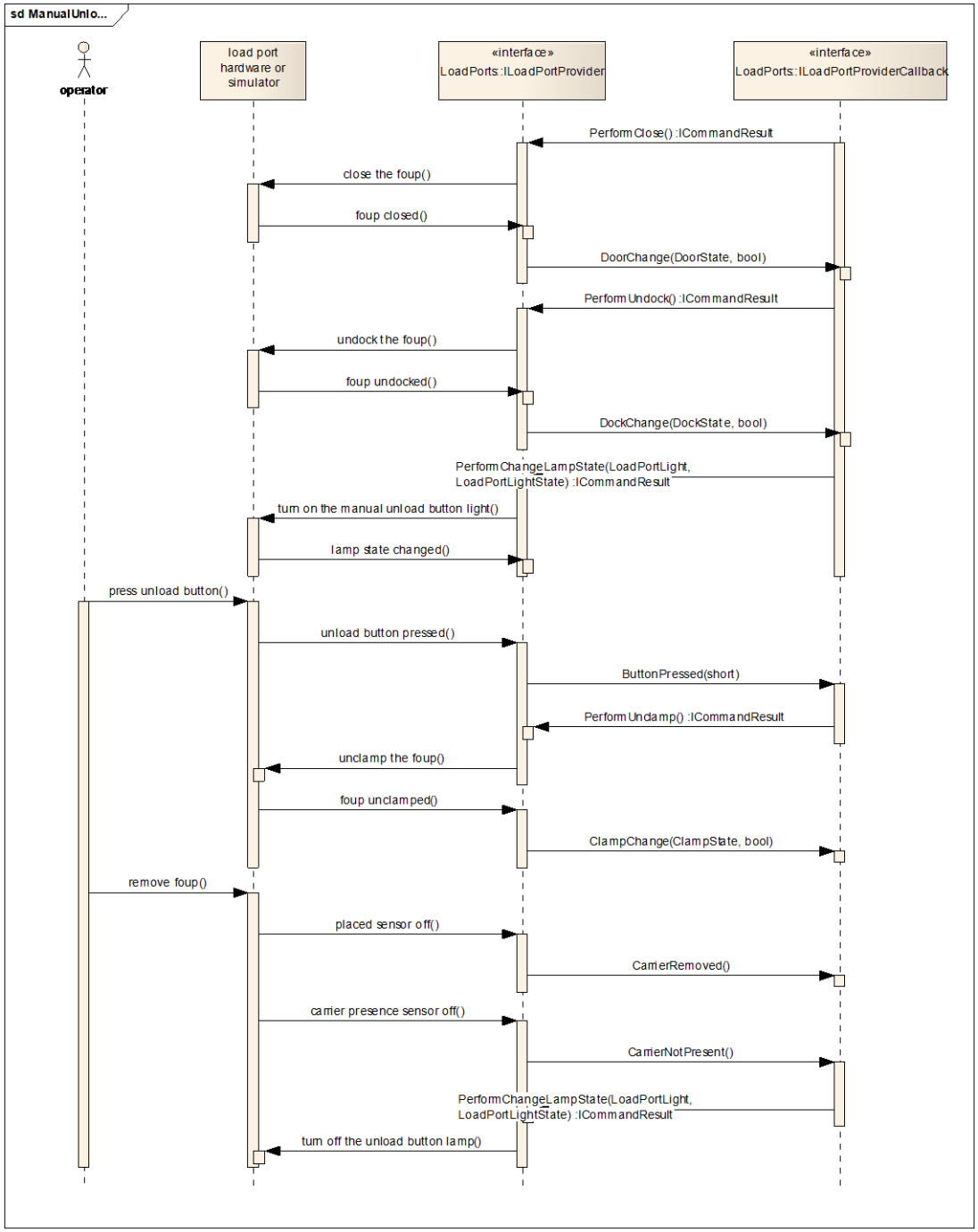
2.6.1.4 Manual load sequence

The following sequence diagram shows the typical interactions between the Operator, the load port hardware, and the LoadPort and LoadPortProvider classes when a FOUP is manually loaded. In this diagram, the LoadPortProvider class is identified by the interface it implements: ILoadPortProvider and the LoadPort class is identified by the interface it implements: ILoadPortProviderCallback.



2.6.1.5 Manual unload sequence

The following sequence diagram shows the typical interactions between the Operator, the load port hardware, and the LoadPort and LoadPortProvider classes when a FOUP is manually unloaded. In this diagram, the LoadPortProvider class is identified by the interface it implements: ILoadPortProvider and the LoadPort class is identified by the interface it implements: ILoadPortProviderCallback.



2.6.1.6 Configuration parameters

Each instance of a load port has an associated set of configuration parameters. The typical location, in the CCF configuration 'tree', for a load port's parameters is ECS.EFEM.LoadPorts.LoadPortX where LoadPortX identifies a particular load port (Ex. LoadPort2). This set of parameters contains a boolean value named "Present" that indicates whether this load port is present in a particular tool's hardware configuration. The load port's configuration parameters may also include parameters that provide the information needed to connect to the load port controller for communication purposes or

parameters that further describe the load port or affect load port behavior. Defining the configuration parameters required to support a particular type of load port is one of the design tasks performed when adding support for a new type of load port.

2.6.1.7 Threading considerations

The methods in the ILoadPortProvider interface (implemented by the LoadPortProvider) will be called on a thread provided by the LoadPort implementation. The LoadPortProvider should not block this thread for long periods of time. The ILoadPortProvider API documentation indicates that all ILoadPortProvider methods return an ICommandResult instance. In the case of 'short duration' load port operations this return value indicates that the requested command has been accepted and is also complete (ex. PerformWriteTagData). In the case of 'long duration' operations, the ICommandResult return indicates that the command has been accepted and that a later status change, indicated by an ILoadPortProviderCallback invocation, marks the completion of the accepted command (ex. PerformDock).

The methods in the ILoadPortProviderCallback interface, called by the LoadPortProvider, are invoked on a thread(s) provided by the LoadPortProvider. There is no guarantee that these method invocations may not block for a significant length of time. If the LoadPortProvider cannot afford for a callback method to block, then the callbacks should be invoked on a separate worker thread. But if worker threads are used, care should be taken that the order of the load port status changes is preserved.

2.6.2 Robot

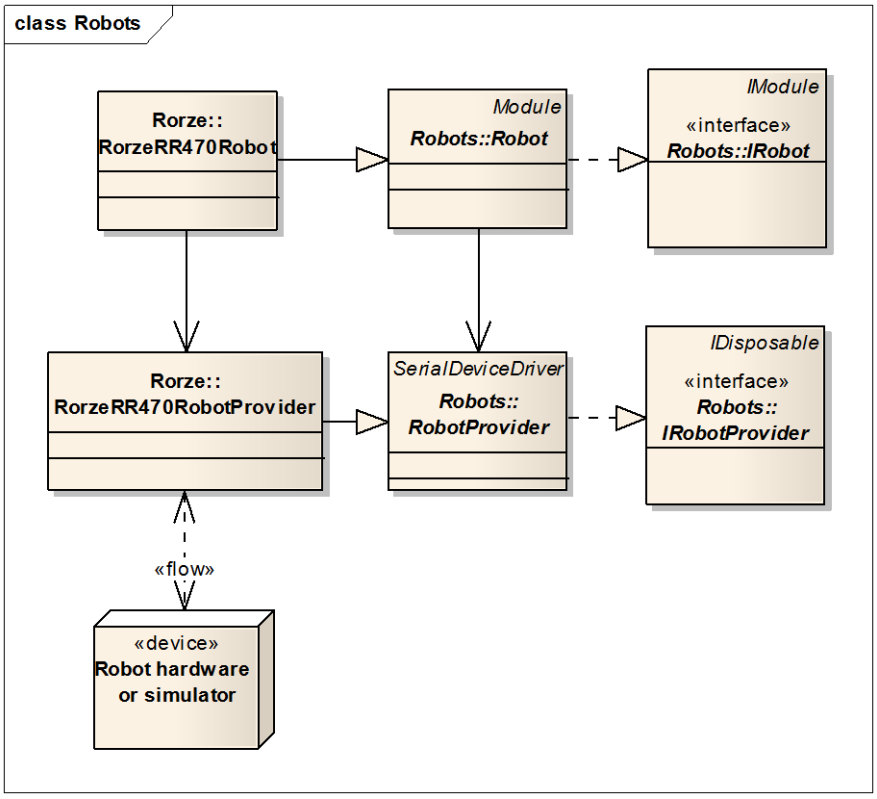
A new type of robot is supported by developing a new robot driver, which consists of two parts:

- A Robot class
- A RobotProvider class

For most drivers, the Robot class will inherit most or all of its required functionality from its CCF-provided superclass, and the bulk of the driver development work will be to create an appropriate RobotProvider class.

Note: In the following description, the terms Robot and RobotProvider are used to refer to the two classes that comprise the robot driver however those classes will have actual names that reflect the type of robot they support. For example, the driver for the Rorze RR470 robot consists of RR470Robot and RR470RobotProvider classes. The driver's Robot class should not be confused with the CCF class named Robot in the Cimetrix.CimControlFramework.Equipment.Components.Robot namespace. When referring to that CCF class, the fully qualified name will be used to avoid confusion.

The following class diagram illustrates the relationships between the robot driver classes and other relevant CCF classes. Note that the Robot class, in this case RR470Robot, 'contains' the RobotProvider class, named RR470RobotProvider. In this diagram, the RR470Robot class invokes robot command methods, such as PerformPick, on the IRobotProvider interface implemented by RR470RobotProvider. The RR470RobotProvider informs RR470Robot of the robot commands through the return values of the IRobotProvider interface methods. In this way, the driver's Robot class issues commands that the driver's RobotProvider communicates to the robot hardware, and the RobotProvider provides command results back to its corresponding Robot class.



The following two sections describe the responsibilities of the driver's Robot and RobotProvider classes in more detail.

2.6.2.1 Robot responsibility

The driver's Robot class is responsible for overall robot behavior and for interfacing to other parts of CCF. This class should be a subclass of the Cimetrix.CimControlFramework.Equipment.Components.Robots.Robot class. The Robot class is also responsible for instantiating and keeping a reference to an instance of the driver's RobotProvider class.

2.6.2.2 Constructor sequence

Typical code for a Robot class constructor is shown below:

```
/// <summary>
/// Contructor
/// </summary>
/// <param name="name">Name of the robot</param>
/// <param name="parameterPath">Path to the configuration parameters</param>
public RorzeRR470Robot(string name, string parameterPath)
: base(name)
{

this.parameterPath = parameterPath;

// Create the robot provider.
// Store the provider in this class and the base class.
```

```
RorzeRR470RobotProvider rProvider = new
RorzeRR470RobotProvider(CxPath.Combine(name, "Provider"), serialConnection);
base.Provider = rProvider;

// Register for updates from provider
rProvider.ToolStatus += OnToolStatus;

//----------------------------------------------------------------------
// Add substrate locations
try
{
    // Register substrate locations
    SubstrateLocationData sd = new SubstrateLocationData(name + ".ArmA",
LocationType.Robot);
    sd.ModuleSlot = 1;
    AddLocation(sd);
    sd = new SubstrateLocationData(name + ".ArmB", LocationType.Robot);
    sd.ModuleSlot = 2;
    AddLocation(sd);
}
catch( Exception exp )
{
    Log.WriteExceptionCatch(Name, exp);
}

// Register data variables to be published
RegisterVariables();

// Start the command processing thread.  This should always be done last.
base.Start();
}
```

The Robot constructor is typically invoked once for each of the tool's robots. One pair of Robot/RobotProvider instances is created for each physical robot present on the tool.

2.6.2.3 ConfigureIO

Often, all of the information needed by the robot comes through the connection to the robot provided by the RobotProvider. If some of the information comes through CCF's IO packages, the the I/O endpoints should be configured by overriding the base class's ConfigureIO method. The OutputEndPoints and InputEndPoints should be created, assigned to IO Groups, and started.

2.6.2.4 GenerateGuiDataVariableNames and GenerateGuiDataValues

The derived Robot class may wish to publish additional data variables. One method of publishing these data variables is to override the GenerateGuiDataVariableNames and GenerateGuiDataValues methods. These methods publish data variables from all ancestor and derived classes at the same time. Publishing these data variables is a two step process.

First, all of the names are generated. This is done once, during startup. The base class calls GenerateGuiDataVariableNames. All the derived classes append the names of the data variables they will be publishing to the GuiDataVariableNames string array, storing the offset into the array where its variables start. Each class must call the base class's GenerateGuiDataVariableNames method.

Second, when the base class PublishGuiData or PublishGuiDataNow methods are called, the base class calls GenerateGuiDataValues. All the derived classes insert the current values of the data variables into the GuiDataValues object array, starting at the stored offset and using the same order that the names were added to the GuiDataVariableNames. Each class should call the base class's GenerateGuiDataVariableValues. Once all the derived classes have inserted the current variable values, the base class calls the FactoryAutomation package method SetVariables with the GuiDataVariableNames array and the newly generated data variable values.

2.6.2.5 RegisterNotificationServiceServices

As the interface between the robot and the rest of CCF, the Robot class can register Notification services that the robot will be providing. Notification services are typically the means used by the Operator interface to provide services. Note that often services that use multiple modules, like wafer transfer commands, are often provided by higher level objects which have visibility into all the modules involved.

2.6.2.6 RobotProvider responsibility

The driver's RobotProvider class is responsible for communicating robot commands to the robot hardware. These commands are invoked on the RobotProvider via methods in the IRobotProvider interface and RobotProvider base class methods.

The RobotProvider is also responsible for collecting up-to-date status from the robot hardware. This status may be collected by polling the hardware and/or by event messages sent by the hardware. When the RobotProvider receives a status update, it updates CCF with the latest status primarily by notifying its Robot class, which then updates the CCF data variables. In some designs, the RobotProvider class updates the CCF data variables directly.

Note: All communication with the robot hardware is handled by the driver's RobotProvider class. This communication is the primary role of the RobotProvider class.

The RobotProvider class is usually not responsible for enforcing substrate transfer sequencing behavior or constraints. For example, when told to 'pick the wafer in LP1 slot 5', the RobotProvider class simply issues the command(s) to pick the wafer to the robot hardware. This will usually involve translating a CCF substrate location (LP1 Slot 5) to a robot station.

Exception: In special cases, the RobotProvider may need to check robot state or the state of an interlock before issuing a command. For example, it's obviously important that the load port door is not closed while a robot is accessing substrates in the FOUP at that load port. If sufficient interlocks are not provided by the hardware or hardware controllers, special handling to prevent this situation may need to be added to the Robot or RobotProvider classes.

Descriptions of the methods defined in the these classes and interfaces can be found in the source code comments.

2.6.2.7 Initialization sequence

When PerformInitialize method is called, the RobotProvider should do the following:

1. Establish the communication connection to the robot hardware (if not already done). This is normally done by the Cimetrix.CimControlFramework.Equipment.Components.Robots.Robot class during its initialization by calling the RobotProvider's Connect method.
2. Attempt to clear any pre-existing error condition that might exist. This is required because re-initializing is sometimes done to try to rectify an otherwise unresolvable error condition.
3. Perform any initialization necessary to insure that the robot is ready to be used and that robot status will be collected and kept current.

4. 'Home' or perform an origin search with the robot mechanism, if applicable.
5. Collect the current status of the robot.
6. Return a command result which indicates if the initialization succeeded or not.

2.6.2.8 Connect sequence

Connecting to a TCP/IP connection is normally handled by the Cimetrix.CimControlFramework.Equipment.Components.Robots.RobotProvider class. However, if a message must be sent to complete the connection sequence, override the Connect method to send the message.

2.6.2.9 ServiceState sequence

The ServiceState method is defined by the ActiveObject class, which is a base class for both Robot and RobotProvider. The ServiceState method gets called periodically (default is 200 ms) when there is not a blocking command being executed. The ServiceState method can be used to monitor connection status by the RobotProvider.

2.6.2.10 Configuration parameters

Each instance of a Robot has an associated set of configuration parameters. The location, in the CCF configuration 'tree', is normally passed into the Robot constructor. The Robot's configuration parameters may include parameters that provide the information needed to connect to the robot controller for communication purposes or parameters that further describe the robot or affect robot behavior. Defining the configuration parameters required to support a particular type of robot is one of the design tasks performed when adding support for a new type of load port. It is recommendation that the translation of CCF substrate locations to robot stations be configurable.

2.6.2.11 Threading considerations

Many methods in the RobotProvider called by the Robot depend on the return code to update the Robot state. This design requires the RobotProvider to block this thread until the command is complete or has failed. Any intermediate updates should be done on a thread provided by the RobotProvider.

Communication from the RobotProvider to the Robot is currently not structured by CCF.

2.7 Creating CIMPortal deployment packages

The procedure below describes how to create a CIMPortal deployment package. A deployment package is used by CIMPortal to represent a logical layout of the equipment. Most parameters, events, and alarms are added to this layout dynamically during equipment startup. The solution describes below is specifically for the CCF sample solution, but it can easily be adapted for a project.

Adding parameters to the model is found in "Creating and updating variables".

Adding events to the model is found in "Creating and using events".

Adding alarms to the model is found in "Creating and using execution alarms", "Creating and using managed alarms", and "Creating and using notification alarms".

2.7.1 Build CIMPortal Model

1. Open CIMPortal's Equipment Model Developer (EM Developer). It can be accessed through the Start menu at Cimetrix | CIMPortal | Equipment Model Developer.
2. Open EM Developer's DCIM Administrator using the Tools | DCIM Administrator menu item.
   a. Click the DCIM Administrator "Load DCIM Instance" button. The "DCIM Instance Selection" dialog should appear.
   b. Verify the EquipmentData and FactoryAutomation DCIMs are present.

c. Click “Cancel” on the “DCIM Instance Selection” dialog.

d. Click “Close” on the DCIM Administrator.

e. If the EquipmentData DCIM was not present, follow the instructions in “Creating the EquipmentData DCIM Instance”.

f. If the FactoryAutomation DCIM was not present, follow the instructions in “Creating the FactoryAutomation DCIM Instance”.

3. Open the CCF solution CIMPortal model file in the Equipment Model Developer. The model file is found at CCF\Source\EquipmentModels\CIMPortal Model\Equipment.xml.
4. Modify the model as necessary. All modules except process modules should be added as “Subsystems”. Remember to change the name when adding new nodes.
5. Do final validation
6. Rebuild the deployment package. The password should be “equipment”. The deployment package should be created at CCF\Source\EquipmentModels\CIMPortal Model\Equipment.pkg.
7. Deploy the new deployment package into CIMPortal. This is done through the start menu at Cimetrix | CIMPortal | CIMPortal Control Panel. The Client Name and Password are both “admin”. Click the Deploy button at the bottom of the control panel. Browse to and open the CCF\Source\EquipmentModels\CIMPortal Model\Equipment.pkg file. Type in the password “equipment” used when the deployment package was built.
8. It may be necessary to restart the CxConfigSvc, CxCIMPortalSvc, and CxCIMStoreSvc services before the changes take effect.

2.7.2 Creating the EquipmentData DCIM Instance

These steps are only needed if the EquipmentData DCIM Instance has not already been created in EMDeveloper. Note: this procedure assumes CIMPortal was installed in the default directory.

1. Create a CCFWCFAppDCIM DCIM Package in CIMPortal
   a. Check the CIMPortal installation to see if the CCFWCFAppDCIM DCIM Package has already been installed. Do this by checking for a folder named C:\Program Files\Cimetrix\Comm Products\bin\Storage\DCIM Packages\CCFWCFAppDCIM. If this folder doesn’t exist, create it.
   b. Copy the following files to the CCFWCFAppDCIM folder from the CCF\Source\Shared folder
      Cimetrix.CxCCFWCFAppDCIM.dll
      Cimetrix.CxCCFWCFAppDCIMClient.dll
      Install_CCFWCFAppDCIM.bat
      Interop.CxAppDCIMLib.dll
      Interop.DCIMInterfaceLib.dll
      Interop.VALUELib.dll
      Uninstall_CCFWCFAppDCIM.bat
   c. Run the Install_CCFWCFAppDCIM.bat batch file.
2. Open EMDeveloper
3. Open the DCIM Administrator using the Tools menu.
4. Put “EquipmentData” in the “Instance Name” field.
5. Select “CCFWCFAppDCIM” in the “DCIM Package” dropdown.
6. Put “Equipment.adc,1,net.tcp://localhost:4866” in the “Model Config String” field.
7. Put “Equipment.adc,1,net.tcp://localhost:4866” in the “Runtime Config” String field.
8. Click the “Add File(s)” button and browse to the Equipment.adc file in the CCF\Source\EquipmentModels \CIMPortal Model\DCIM Instances\EquipmentData folder and add it to the instance.
9. Click the “Save DCIM Instance” button

2.7.3 Creating the FactoryAutomation DCIM Instance

These steps are only needed if the FactoryAutomation DCIM Instance has not already been created in EMDeveloper.

1. Open EMDeveloper
2. Open the DCIM Administrator using the Tools menu.
3. Put "FactoryAutomation" in the "Instance Name" field.
4. Select "CIMConnectDCIM" in the "DCIM Package" dropdown.
5. Put "localhost,0,3,1,20000" in the "Model Config String" field.
6. Put "localhost,0,3,1,20000" in the "Runtime Config" String field.
7. Click the "Save DCIM Instance" button

2.8 Creating light tower conditions

2.8.1 Overview

The light tower uses variable values from CIMConnect to evaluate conditions. The evaluation results are used to determine the correct state for each signal controlled by the light tower. These conditions are compiled into C# code and are evaluated when CIMConnect detects a change in a variable value.

The behavior of the light tower can be modified by changing the conditions of its signals. These conditions are defined in the configuration file. Conditions can be added, deleted, or modified to meet the needs of each project.

Each signal may have three states: Off, On, and Blink. A buzzer signal may only have two states: Off and On. Each signal state has an associated condition. As GEM variable values change, alarms are set and cleared, and events are triggered, the conditions for each signal state are evaluated. Depending on the evaluation results, the signal states are updated.

The signal state conditions are evaluated in the following order:

- Off
- On
- Blink

The last true condition will be the new signal state. If all conditions evaluate to be true, the signal will be in the Blink. Conversely, if all conditions are false, the signal is left in its previous state.

There are generally two styles of conditions: state-driven and event-driven. State-driven conditions are constructed from values which persist, like parameter values and alarm states. Event-driven conditions are triggered from GEM events. State-driven conditions and event-driven conditions generally should not be used in the same signal.

The most common state-driven condition for the Off state is "true".

2.8.2 Writing Conditions

A condition is simply an equation that evaluates to a Boolean (true or false) value. A condition may use GEM parameters, events, and alarms. Comparisons may be made with C# relational operators (==, !=, <, >, <=, >=). The results of comparisons may be combined using C# logical operators (!, &&, ||, ^) and nested using parenthesis. Conditions may be as simple or complex as needed. The constants "true" and "false" may be used.

A GEM parameter value may be used by placing GetParameterValue ("parameterName") in the condition where parameterName is replaced by the name of the parameter.

A GEM alarm may be used by placing IsAlarmSet (“alarmName”) in the condition where alarmName is replaced by the name of the alarm.

A GEM event may be used by placing IsEventTriggered (“eventName”) in the condition where eventName is replaced by the name of the event.

Examples:

A condition that is true if loadport1, loadport2, or loadport 3 is ready to unload

GetParameterValue("PortTransferState_1") == 3 || GetParameterValue("PortTransferState_2") == 3 || GetParameterValue("PortTransferState_3") == 3

A condition that is true if the process state of the tool is executing

GetParameterValue("PROCESSSTATE") == 4

A condition that is true when material is received at the tool

IsEventTriggered(“MaterialReceived”)

A condition that evaluates true when loadport1 has a carrier present or carrier placement error

IsAlarmSet(“LP1.CarrierPresentError”) || IsAlarmSet(“LP1.CarrierPlacementError”)

2.8.3 Relational operators

The following relational operators are supported:

Operator	Description
==	The equality operator returns true if the values of its operands are equal, false otherwise
!=	The inequality operator returns true if the values of its operands are different, false otherwise
<	The “less than” operator returns true if the first operands is less than the second, false otherwise
>	The “greater than” operator returns true if the first operands is greater than the second, false otherwise
<=	The “less than or equal to” operator returns true if the first operands is less than or equal to the second, false otherwise
>=	The “greater than or equal to” operator returns true if the first operands is greater than or equal to the second, false otherwise

2.8.4 Logical operators

The following logical operators are supported:

Operator	Description
!	The logical negation operator negates its operand. It returns true if and only if its operand is

	false.
&&	The conditional-AND operator performs a logical-AND. It returns true if and only if both operands are true.
||	The conditional-OR operator performs a logical-OR. It returns true if either or both operands are true.
^	The conditional-XOR operator performs a logical exclusive OR. It returns true only if one operand is true and the other is false.

2.8.5 GetParameterValue

GetParameterValue is the method by which GEM parameter values can be used by a condition. It takes a single argument which is a string containing the name of the parameter. It returns a double representing the value of the parameter. In C#, it would be defined as:

double GetParameterValue(string name)

A double cannot be implicitly converted to a Boolean value, so a condition which uses GetParameterValue must use a relational operator to explicitly convert the parameter value into a Boolean result. Once converted, it can be directly used as the result of the condition or it may be combined with other Boolean values using logical operators.

For A or W type parameters, GetParameterValue returns the length of the text.

For L type parameters, GetParameterValue returns the number of items in the list.

2.8.6 IsAlarmSet

IsAlarmSet is the method by which GEM alarm states can be used by a condition. It takes a single argument which is a string containing the name of the alarm. It returns a Boolean representing the state of the alarm, true if the alarm is set, otherwise false. In C#, it would be defined as:

bool IsAlarmSet(string name)

The return value of IsAlarmSet can be directly used as the result of the condition or it may be combined with other Boolean values using logical operators.

2.8.7 IsEventTriggered

IsEventTriggered is the method by which GEM events can be used by a condition. It takes a single argument which is a string containing the name of the event. It returns a Boolean representing if the event is currently triggered, true if the event is triggered, otherwise false. In C#, it would be defined as:

bool IsEventTriggered(string name)

The return value of IsEventTriggered can be directly used as the result of the condition or it may be combined with other Boolean values using logical operators. When designing event-based conditions, it is important to remember that events are only triggered momentarily.

2.9 Integrating a new process module using the ProcessModule base class

2.9.1 Overview

For developers who wish to integrate a new process module into the CIMControlFramework™ equipment control layer, using the ProcessModule base class may be the fastest solution. This procedure does not apply to process modules which contain more than one substrate per chamber. This type of process module will need to be developed as a purely custom class and not derived from the ProcessModule class.

2.9.2 Description

- Start a new .NET framework solution.
- Add a reference to the CCF Equipment package. It is contained in the Cimetrix.Equipment.dll assembly. You may need to add references to other CCF packages, such as Logging. If you need to look at the source code, the ProcessModule class is found in CCF\Source\Equipment\Modules\ProcessModule.cs. You may also wish to reference the DummyProcessModule class in CCF\Source\Equipment\Modules\DummyProcessModule.cs. The DummyProcessModule class is used in the CCF sample solution.
- Add a new class which derives from the ProcessModule class. For purposes of this procedure, the name "MyProcessModule" will be used for this new class.
- Implement a class constructor
    a. The constructor needs to call the base class constructor with a string name. The string name is normally passed into the constructor. If you want the ProcessModule class to control the gate valve, the interface to the gate valve must be passed into the base class constructor.
    b. For each chamber on the process module, call ProcessModule.AddChamber. Each processing chamber may contain a single substrate.
    c. Read the value of any configuration parameters that the process module might use.
    d. Register any dynamically-defined parameters, as described in Section 00.
    e. Call the ActiveObject.Start method. This starts the ActiveObject thread.
- Implement abstract methods
    a. PerformPrepareForRecipes—Prepare for processing a wafer with a given recipe. It is not expected to return until the preparation is complete or aborted. This method can be stubbed out for mechanical cycling. The module state should be updated at the beginning and end of this method. For example:

```
SetModuleState(ModuleStateEnum.Performing);

// Prepare here

SetModuleState(ModuleStateEnum.Idle);
```

    b. PerformProcess—Process the given wafer with the given recipes. It is not expected to return until the process is completed or aborted. This method can be stubbed out for mechanical cycling. The module state should be updated at the beginning and end of this method. For example:

```
SetModuleState(ModuleStateEnum.Processing);

// Process here

SetModuleState(ModuleStateEnum.Idle);
```

c. AbortAllProcess—Signal all chambers that any Process commands should be aborted. Process commands are free to ignore an abort request. This command should return as soon as the signal has been sent and should not wait until the abort is completed. This method can be stubbed out for mechanical cycling.

d. AbortProcess—Signal a single chamber that its Process command should be aborted. If appropriate, this method can also call AbortAllProcess. This method can be stubbed out for mechanical cycling.

e. AbortAllPrepareForRecipe—Signal all chambers that any PrepareForRecipe commands should be aborted. PrepareForRecipe commands are free to ignore an abort request. This command should return as soon as the signal has been sent and should not wait until the abort is completed. This method can be stubbed out for mechanical cycling.

f. AbortPrepareForRecipe—Signal a single chamber that its PrepareForRecipe command should be aborted. If appropriate, this method can also call AbortAllPrepareForRecipe. This method can be stubbed out for mechanical cycling.

g. PerformOpenGateValve-Opens the gate valve. This command is not expected to return until after the gate valve is open. This method can be stubbed out for mechanical cycling.

h. PerformCloseGateValve- Closes the gate valve. This command is not expected to return until after the gate valve is open. This method can be stubbed out for mechanical cycling.

i. IsRecipeSetValid—Determine if a set of recipes is valid. The validity is determined solely by the process module. This method can be stubbed out (always return true) for mechanical cycling.

j. IsPreparedForRecipes—Determine if the process module is ready to process a set of recipes. The readiness is determined solely by the process module. This method can be stubbed out (always return true) for mechanical cycling.

- Override methods which need custom functionality.

a. ServiceState—Called periodically by the ActiveObject thread (when not processing a command) so that states may be updated or IO sensor changes can be evaluated. Also typically used to publish non-process related data variables.

b. GenerateGuiDataVariableNames—Called once during ActiveObject startup to generate the data variable names that will be published when PublishGuiData or PublishGuiDataNow is called. This data will include data variables from all parent and derived classes.

c. GenerateGuiDataValues—Called during PublishGuiData or PublishGuiDataNow to get the data values to be published. The data values must be placed at the indices corresponding to the data variable names.

d. PerformPrepareForTransfer—Prepare for a substrate transfer with a robot. This command is not expected to return until the process module is ready for the transfer. Once this command returns, the robot will extend into the process module for substrate transfer, so all gate valves must be opened, lift pins moved to the correct transfer position, etc. This method must be should be overridden before mechanical cycling.

e. PerformNoteTransferInProgress—Notification that a robot has started substrate transfer with the process module. In normal operation, a PerformPrepareForTransfer will precede this call. This method is expected to return quickly.

f. PerformNoteTransferComplete—Notification that a robot has completed substrate transfer with the process module. In normal operation, a PreformNoteTransferInProgress will precede this call. This method is expected to return quickly.

g. PerformInitialize—Called by the ActiveObject thread when the process module is being initialized. Subcomponents should also be initialized. The module state should be set

to “Initializing” at the beginning of initialization and updated again at the end. For example:

```
SetModuleState(ModuleStateEnum.Initializing);
// Initialize here
ModuleStateEnum state = (succeeded) ? ModuleStateEnum.Idle :
ModuleStateEnum.NotInitialized;
SetModuleState(state);
```

- Modify the application which constructs the equipment control layer to instantiate the MyProcessModule class. If you are using the ToolSupervisor provided in the sample CCF solution, make a backup copy before modifying the sample source code.
- To add alarms, see Samples section under the Alarm Package section of this document.
- To add configuration parameters, see “TBD-Configuration”.
- To add new data variables, see “Dynamically Adding Parameters to a Process Module”.
- To add logging use the various methods on the Log class to post logging messages.

2.10 I/O Configuration

The mapping of CCF I/O endpoints to hardware I/O channels is controlled by I/O configuration files.

The I/O configuration file name is determined by two items: the ConfigurationClient.BaseName, and whether the application is configured to use real hardware or simulated hardware. A configuration parameter named ConfigurationClient.BaseName + “.UsingRealHardware” is used to determine if the application is to use real hardware or not. The sample CCF solution uses the configuration parameter ToolSupervisor.UsingRealHardware. If the application is using real hardware, the I/O configuration file is named ConfigurationClient.BaseName + “IOConfiguration.xml”. If the application is using simulated hardware, the I/O Configuration file is named ConfigurationClient.BaseName + “SimIOConfiguration.xml”. These files should be located in the startup directory, which is typically the directory where the application is located.

Note that the standard CCF Simulated IO Provider does not require an I/O configuration file.

A portion of the IO configuration file is common for all types of I/O hardware, but most of the format is determined by the software responsible for mapping the CCF IO endpoints to the hardware I/O channels. This software is called the IOProvider. CCF has built-in support for three types of IOProviders—Modbus, OPC, and Simulated. Additional IOProviders can be created without changing other software components.

2.10.1 Modbus I/O Configuration file

A sample Modbus I/O configuration file named “SampleModbusIOConfiguration.xml” has been included in the documentation folder.

The first section needed by the CCF Modbus I/O layer is the group section, which is common to all IOProviders. This section looks like this:

```
<!--
=====================================================
= Scanned I/O Groups Configuration.
=====================================================
-->

<Groups>
  <Group Name="LL1.Door.IOGroup" Target="modbus-tcp-server:1"/>
  <Group Name="LL1.GV.IOGroup" Target="modbus-tcp-server:1"/>
```

```
    </Groups>
```

This sections contains a mapping of the all the I/O groups that will be present in the solution to the IOProvider for that group. This mapping allows multiple types of underlying hardware to be in the solution at the same time. It also requires all the I/O endpoints in a single group to be supported by the same IOProvider. The Target attribute determines the type of IOProvider that will be instantiated for the I/O group whose name matches the Name attribute. In this case, these groups will both use the Modbus server with Id of "1".

The remaining configuration for the Modus I/O layer is contained in a Modbus specific section. The first part of this section defines the number of Modbus servers in the solution. This section looks like:

```
      <Servers>
        <Server ID="0" Address="172.20.9.0" Port="502" RequestTimeout="10000"
MaxRetry="0" MaxExceptionBeforeReset="5" EstablishCommunicationTimeout="10000"
BurstSize="1"/>
        <Server ID="1" Address="172.20.9.1" Port="502" RequestTimeout="10000"
MaxRetry="0" MaxExceptionBeforeReset="5" EstablishCommunicationTimeout="10000"
BurstSize="1"/>
      </Servers>
```

The next section contains the read schemes.

```
      <ReadSchemes>
        <Scheme GroupName="1">
          <Request ServerID="1" ReadFunction="2" StartAddress="0" Count="92">
          </Request>
        </Scheme>
      </ReadSchemes>
```

Each read scheme belongs to a Modus server, and defines the Modbus function to be used, the starting address, and the number of addresses to be read. This read scheme allows multiple I/O groups to be read in one Modbus request. Ideally, there should be only one read scheme per server.

The next section contains the input (sensor) mappings.

```
      <Mapping>
        <Inputs>
          <Input Tag="LL1.Door.iIsOpened" ServerID="1" RequestIndex="0" Offset="52"
/>
          <Input Tag="LL1.Door.iIsClosed" ServerID="1" RequestIndex="0"
Offset="53"/>
          <Input Tag="LL1.GV.iIsOpened" ServerID="1" RequestIndex="0" Offset="54"/>
          <Input Tag="LL1.GV.iIsClosed" ServerID="1" RequestIndex="0" Offset="55"/>
        </Inputs>
      </Mapping>
```

Each input endpoint in the solution using a Modbus IOProvider should have an input mapping. This will map the endpoint value to a location in the previously defined read schemes.

NOTE: By changing the Offset and/or Request Index in the input sensor mapping, the value of an Input Endpoint may be mapped to a different sensor location without requiring a code change.

The final section contains the write schemes.

```
      <WriteSchemes>
```

```
        <Scheme GroupName="LL1.Door.IOGroup">
          <Request ServerID="1" WriteFunction="6" StartAddress="2051" Count="1">
            <Map Tag="LL1.Door.oSetOpen" Offset="0" />
            <Map Tag="LL1.Door.oSetClose" Offset="1" />
          </Request>
        </Scheme>
        <Scheme GroupName="LL1.GV.IOGroup">
          <Request ServerID="1" WriteFunction="6" StartAddress="2051" Count="1">
            <Map Tag="LL1.GV.oSetOpen" Offset="2" />
            <Map Tag="LL1.GV.oSetClose" Offset="3" />
          </Request>
        </Scheme>
      </WriteSchemes>
```

Each write scheme belongs to both a Modus server and an IOGroup. It defines the Modbus function to be used, the starting address, and the number of addresses to be written. It also includes the mapping of each output endpoint belonging to the IOGroup to a location in the write scheme.

NOTE: By changing the Offset and/or StartAddress in the write scheme, the value of an Output Endpoint may be mapped to a different actuator (coil) location without requiring a code change.

2.10.2 OPC I/O Configuration file

A sample OPC I/O configuration file named “SampleOPCIOConfiguration.xml” has been included in the documentation folder.

The first and only section needed by the CCF OPC I/O layer is the group section, which is common to all IOProviders. This section looks like this:

```
    <!--
    ====================================================
    = Scanned I/O Groups Configuration.
    ====================================================
    -->

    <Groups>
      <Group Name="LL1.Door.IOGroup" Target="opc-da-server:{a1000002-5694-495c-
b465-51925ed9fbbc}"/>
      <Group Name="LL1.GV.IOGroup" Target="opc-da-server:{a1000002-5694-495c-b465-
51925ed9fbbc}"/>
    </Groups>
```

This sections contains a mapping of the all the IOGroups that will be present in the solution to the IOProvider for that group. This mapping allows multiple types of underlying hardware to be in the solution at the same time. It also requires all the I/O endpoints in a single group to be supported by the same IOProvider. The Target attribute determines the type of IOProvider that will be instantiated for the IOGroup whose name matches the Name attribute. In this case, these groups will both use the OPC server with URL of {a1000002-5694-495c-b465-51925ed9fbbc}. The names of the group’s Input and Output Endpoints are used as the OPC tags.

CCF does not require any additional configuration for OPC support, but the OPC servers must be configured to map the tag names to sensors and actuators.

NOTE: Mapping OPC tag names to different sensors and actuators must be done in the OPC servers.

2.10.3 Extra I/O Channels

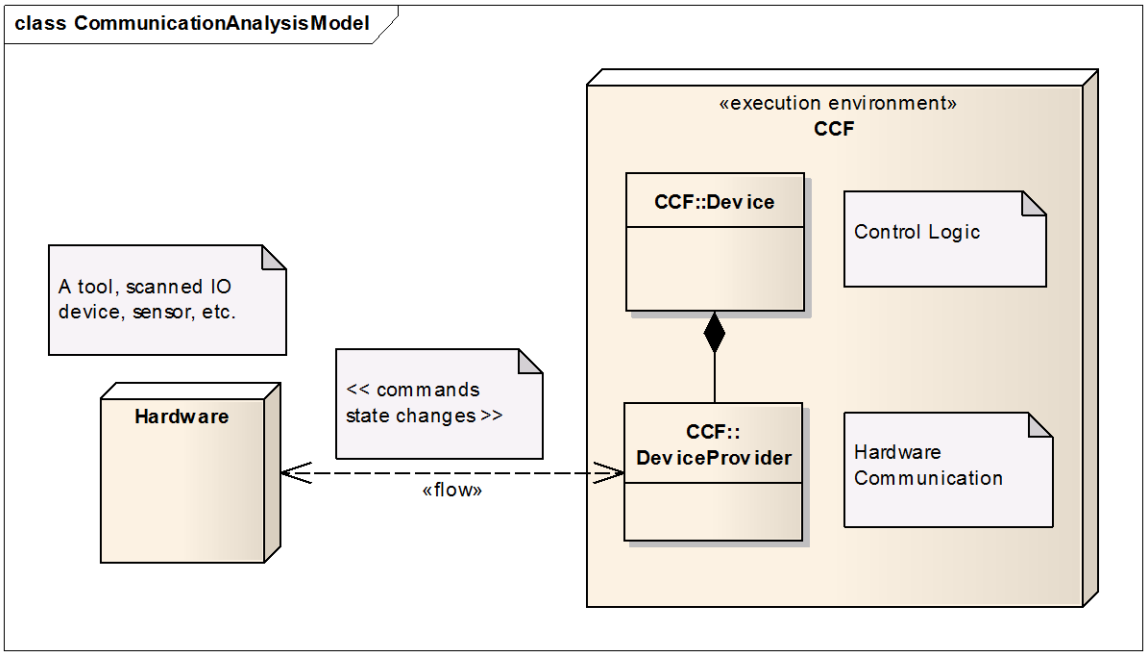
Any extra I/O channels should be mapped and their values published. This makes it easy for the factory to install devices on those channels and collect the data through either the GEM or Interface A connections.

2.10.4 Swapping an I/O Channel

With an IOProvider that uses a mapping scheme such as those described above, it makes it easy for a field service engineer to swap a sensor from a dead channel to a good channel without having to rebuild the software. The engineer need only modify the mapping file to move the signal to the good channel and restart the software.

2.11 Device IO

To control a hardware device or tool, CCF uses a separation of concerns approach to the equipment control design. The first concern is control logic which is the responsibility of an equipment control device object. The second concern is hardware communication, which is the responsibility of the equipment control device provider. It is this second concern of communicating with the hardware that is covered in this section. This broad concept is illustrated in the model below.



To support IO communication with the hardware, CCF provides the IO package and 2 related packages - IOProvider_Modbus_TCP and IOProvider_OPC_Client_Softing. To use the entities defined in these packages you will need to add a reference to Cimetrix. IO.dll and perhaps Cimetrix.IOProvider_Modbus_TCP.dll and Cimetrix.IOProvider_OPC_Client_Softing.dll, to your equipment control project.

2.11.1 Overview

There are two broad categories of IO device communication – serial and scanned.

2.11.1.1 Serial Device IO Overview

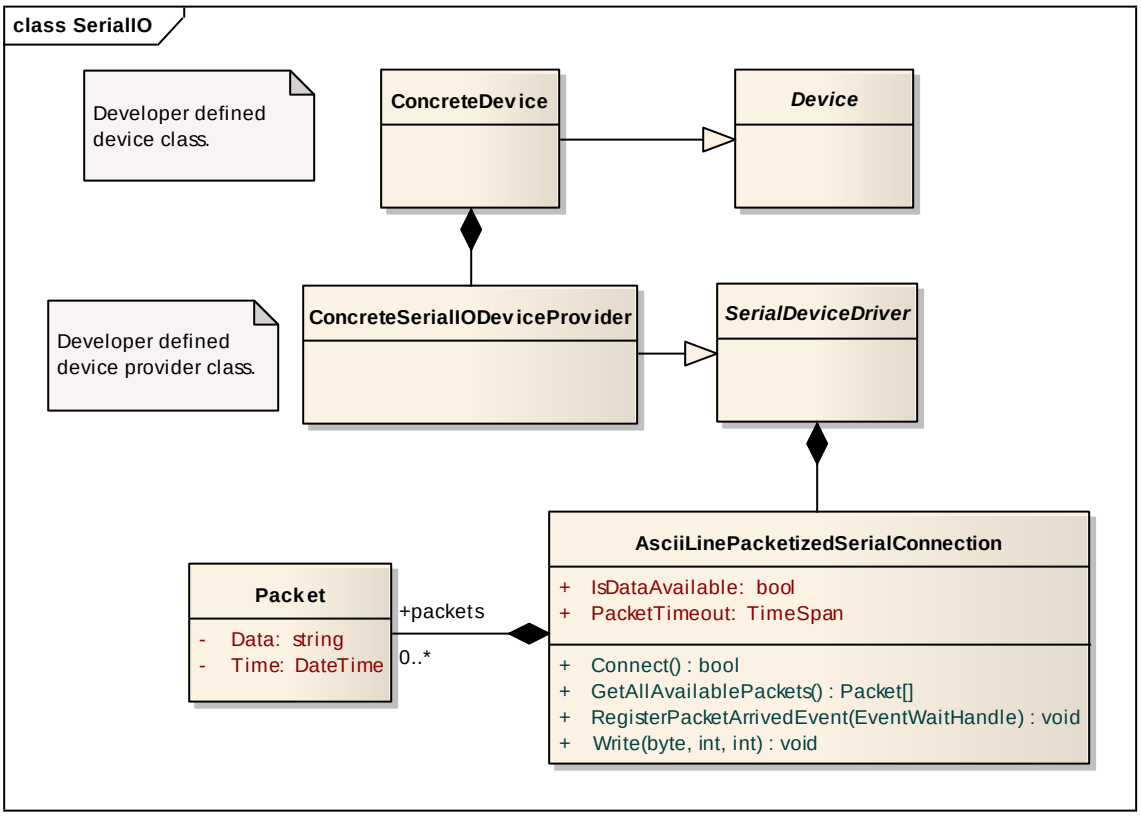
Serial device IO involves a simple client server model with two-way communication. In this scenario, the hardware device functions as the server, waiting to receive commands from a CCF equipment control object, which acts as the client. The hardware also sends information, such as a change in tool state or a notification or alarm, back to the equipment control client object unsolicited. The communication protocol is TCP over a socket.

2.11.1.2 Scanned Device IO Overview

Scanned device IO involves a group of IO endpoints where each endpoint in the group has an input value and an output value. The CCF equipment control device provider object sends commands to the scanned IO device by setting the output values of the group and the tool then reads these output values from the scanned IO device and reacts accordingly. Conversely, the tool can set the input values of the endpoint group to indicate changes in tool state, alarms, etc., and the CCF equipment control object then reads in these input values from the scanned IO device. CCF provides several specific implementations of scanned IO depending on the type of device used such as Modbus or OPC.

2.11.2 Using Serial Device IO

The primary IO class used in serial device IO communication is AsciiLinePacketizedSerialConnection. It is found in the Cimetrix.CIMControlFramework.IO.Serial.Packetizers namespace. An object of this type is used by an equipment control device provider object (e.g., a LoadPortProvider object) to send commands to a piece of hardware or tool, and to receive changes in state, notifications, and alarms from that hardware or tool.



Most serial IO provider classes inherit from the abstract class SerialDeviceDriver which encapsulates the AsciiLinePacketizedSerialConnection object and all communication functionality. This keeps the details

of serial IO communication, including the AsciiLinePacketizedSerialConnection object, hidden from the developer of the concrete provider class (see figure above). It is only necessary that the developer provide the hardware's network address and understand the functionality of SerialDeviceDriver class, which is part of the Equipment package, and not IO package discussed in this section. There are cases, however, when the developer may want to bypass inheriting SerialDeviceDriver and use the IO package's AsciiLinePacketizedSerialConnection directly.

2.11.2.1 Construction

There are three AsciiLinePacketizedSerialConnection constructors. Depending on the constructor used, there are up to four parameters that can be supplied.

Parameter	Required	Description
string connectionName	Yes	The logical name of the connection and this used to find the connectionTarget (see below) in the config file if connectionTarget is not otherwise provided.
string connectionTarget	No	The protocol and address of the listening hardware or tool. A typical value is "tcp://127.0.0.1:10050".
int maxPacketLength	No	Used to limit the length of ASCII lines, default is 512, minimum is 16.
TimeSpan packetTimeout	No	The max time spent waiting for the completion of a packet of data that is being read from the tool, default value is 0.5 seconds.

Once constructed, the AsciiLinePacketizedSerialConnection is still not active, that is, it has not connected to a hardware device and is not currently in communication with anything. The Connect() method must be called to connect to the listening tool and start up communication.

2.11.2.2 Sending Commands to the Hardware

Use the Write() method to send a command to the tool. The Write() method executes on the current thread. The command (message content) must be ASCII text ending with either "\r" or "\r\n". The following is an example of sending a command to the tool.

```
byte[] writeBuffer = Encoding.ASCII.GetBytes(message + "\r\n");
int bytesToWrite = writeBuffer.GetLength(0);
int bytesWritten = serialConnection.Write(writeBuffer, 0, bytesToWrite);
```

Note that the actual interpretation and formatting of this message as a tool command is of no interest to the AsciiLinePacketizedSerialConnection object – it only sees the message as arbitrary text. It is up to the hardware receiving the command and the equipment control provider object issuing the command to agree on the format and interpretation of the text message.

2.11.2.3 Receiving Information Back from the Hardware

The AsciiLinePacketizedSerialConnection class defines a background thread for reading information sent back from the tool. Like the written commands, it expects this information to be ASCII lines ending with either "\r" or "\r\n", but other than this requirement, it sees this information as arbitrary text. For each text line received, a Packet object is created containing the text message

and time of arrival. These packet objects are then placed in a queue according to their time of arrival.

These packets of tool information can be accessed at any time using GetAvailablePacket() which pops the oldest Packet off of the queue and returns it to the caller.

```
Packet packet;
asciiLinePacketizedSerConn.GetAvailablePacket(out packet);
string content = packet.Data;
```

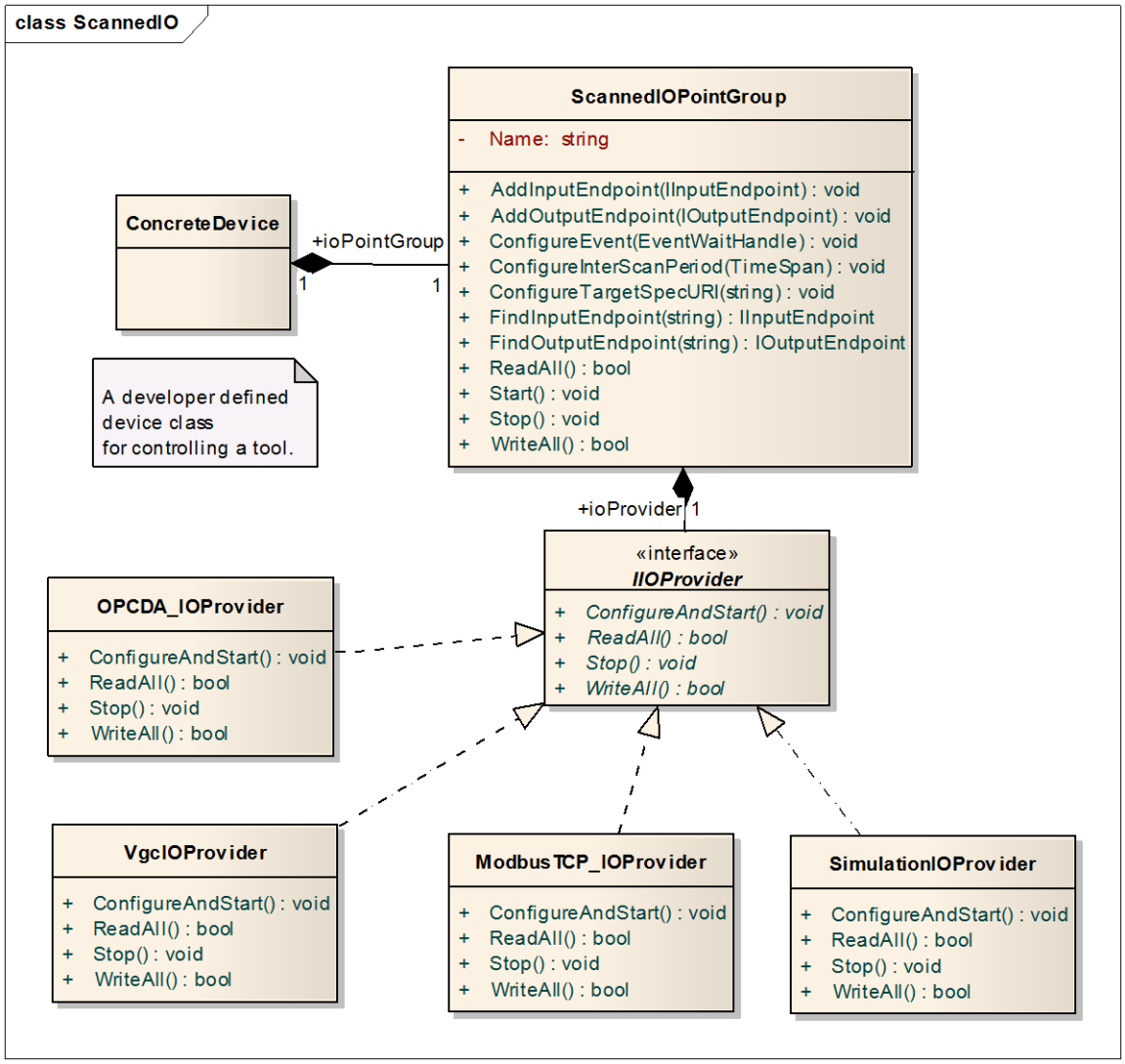
There are two ways that the provider object detects that there are packets in the queue ready to be accessed. First, the AsciiLinePacketizedSerialConnection.IsDataAvailable property indicates if there are, or are not, packets in the queue. Second, a wait handle can be supplied to the AsciiLinePacketizedSerialConnection using the RegisterPacketArrivedEvent() method and this wait handle will be signaled whenever packets are read from the tool. This relieves the equipment control object from having to continually poll to see whether or not the tool has sent back any data. But in order to use this technique, there must be a thread blocking on the wait handle and waiting for the signal.

2.11.3 Using Scanned Device IO

As mentioned previously, equipment control device objects use a device provider object for communicating with the hardware or tool. For scanned IO, the communication is with the scanned device and not the actual tool. Since there are only a few types of scanned devices, CCF has already implemented these provider classes and the developer does not need to (see figure below). This is in contrast to serial device IO mentioned above where the developer must write a concrete provider class for each tool.

Each of the concerete scanned IO provider classes implements the IIOProvider interface. These concrete implementations are VgcIOProvider, ModbusTCP_IOProvider, OPCDA_IOProvider, and SimulationIOProvider.

The device class implementation will not use one these provider classes directly. Instead, the device class implementation will use ScannedIOPointGroup class – an implementation of IScannedIOPointGroup interface and a wrapper for the IIOProvider mentioned above.



2.11.3.1 Device Construction

In the constructor for the device class, you need to include creating and initializing the ScannedIOPointGroup object and this involves several steps. The constructor will most likely contain code for other purposes as well. And, of course, some or all of this code can be factored out into smaller methods called by the constructor.

First create the ScannedIOPointGroup object and assign it to a field or property. The ScannedIOPointGroup constructor takes the tool name as a constructor parameter.

```
ioPointGroup = new ScannedIOPointGroup("TheToolName");
```

Next, configure the endpoint inputs and outputs. In the example below, this is a group of 2 endpoints, each endpoint containing an input and output value. In this example, the endpoints control opening and closing a valve.

```
// first endpoint, valve is open
ioPointGroup.AddInputEndPoint("iepIsOpen", new InputEndPoint<bool>(false));
```

```
ioPointGroup.AddOutputEndPoint("oepSetOpen", new OutputEndPoint<bool>(false));

// second endpoint, valve is closed
ioPointGroup.AddInputEndPoint("iepIsClosed", new InputEndPoint<bool>(false));
ioPointGroup.AddOutputEndPoint("oepSetClose", new OutputEndPoint<bool>(false));
```

Then, tell the ScannedIOPointGroup where to find the scanned device. The value of this parameter will most likely come from the configuration, here it is written as a literal for illustration. Realize that it is this value that determines the type of IIOProvider implementer that is created by the ScannedIOPointGroup object.

```
ioPointGroup.ConfigureTargetSpecURI("modbus"-tcp-server:1");
```

Optionally, pass the ScannedIOPointGroup object the device object's wait handle so that the ScannedIOPointGroup object can notify the device object when inputs have been changed by the hardware and are ready to be accessed.

```
ioPointGroup.ConfigureEvent(ThreadEvent);
```

As another option, you may want to set the scan period. This tells the IO provider object contained within the ScannedIOPointGroup object how often to check for changes to inputs coming back from the hardware.

```
ioPointGroup.ConfigureInterScanPeriod(TimeSpan.FromSeconds(0.2));
```

And finally, start both the ScannedIOPointGroup object and the device object's service thread in that order.

```
ioPointGroup.Start();
Start();
```

After construction, the equipment control device object is ready to begin sending commands to the hardware and is polling for information sent back from the hardware.

2.11.3.2 Sending Commands to the Hardware

To issue commands to the scanned IO device (to then be picked up by the tool), the device object must set values of output endpoints contained in its ScannedIOPointGroup object (created in the constructor above) and then tell the ScannedIOPointGroup object to write those values, using it internal IIOProvider object, to the scanned IO device listening on the other end. An example of this code is shown below.

```
ioPointGroup.FindOutputEndPoint("oepSetOpen").ValueObj = true;
ioPointGroup.FindOutputEndPoint("oepSetClosed").ValueObj = false;
ioPointGroup.WriteAll();
```

2.11.3.3 Receiving Information Back from the Hardware

As an active object (all device classes inherit from ActiveObject), the device object has a worker thread that is constantly looping on a timed interval. Within this loop, the ServiceState() method is called. The concrete device class can override this method to check for, and read, any information sent back from the scanned IO device. A typical example is shown below.

```
if( ioPointGroup.ReadAll() )
{
   iepIsOpen = (bool)ioPointGroup.FindInputEndPoint("iepIsOpen").ValueObj;
   iepIsClosed = (bool)ioPointGroup.FindInputEndPoint("iepIsClosed").ValueObj;
}
```

Also recall that in the device class constructor described previously, the active object wait handle can be passed to the ScannedIOPointGroup object. If this is the case, the ScannedIOPointGroup object will signal this wait handle whenever new information arrives from the IO device, thus short ciruiting the loop's timed interval and calling ServiceState() immediately.

2.11.4 Simulating a Tool Connected to a Scanned IO Device

In order to write a software simulation of a tool that uses a scanned IO device, use the ScannedIOServer in the Cimetrix.CimControlFramework.IO.Scanned.Simulation namespace. This is a WCF service that will listen for changes to scanned IO group outputs and write back tool changes to group inputs. In this way, the ScannedIOServer acts just like a scanned IO hardware device from the point of view of the equipment control provider object.

2.11.4.1 Starting the Server

The ScannedIOServer is a singleton, accessed using the class property Instance. The one instance of this server must be started upon startup of the tool simulation program using the statement below.

```
SimulationIOServer.Instance.StartServiceHost();
```

A good place for this line of code is in the Program.Main() method. Furthermore, the simulation program must provide proper WCF configuration values in its application configuration file. This will look like the following with proper modification to the address attribute of the endpoint element.

```
<system.serviceModel>
  <bindings>
    <netTcpBinding>
      <binding name="CcfDefaultBinding" transferMode="Buffered"
          maxConnections="30" maxReceivedMessageSize="2147483647"
          receiveTimeout="Infinite">
        <readerQuotas maxArrayLength="2147483647"
          maxStringContentLength="2147483647" />
        <security mode="None">
          <transport clientCredentialType="None" protectionLevel="None" />
          <message clientCredentialType="None" />
        </security>
      </binding>
    </netTcpBinding>
```

```
    </bindings>
    <services>
      <service
name="Cimetrix.CimControlFramework.IO.Scanned.Simulation.SimulationIOServer">
        <endpoint address="net.tcp://localhost:6415" binding="netTcpBinding"
            bindingConfiguration="CcfDefaultBinding"
contract="Cimetrix.CimControlFramework.IO.Scanned.Simulation.ISimulationIO" />
      </service>
    </services>
  </system.serviceModel>}
```

This one instance of the server supports all tool simulation objects in the simulation program that use scanned IO. The server distinguishes requests for the various tool simulation objects by tool name. Each service request is accompanied by the name of the tool for which it is intended and in this way, SimulationIOServer routes it to the correct tool simulation object. More on this below.

2.11.4.2 Registering with the Server

Each tool simulation class that implements scanned IO must implement ISimulationIO and register itself with the ScannedIOServer singleton. This registration is done in the tool simulation class constructor and an example is shown below.

```
SimulationIOServer.Instance.RegisterIOProvider("ToolName.p");
```

To the ScannedIOServer, each simulation object is just an implementation of ISimulationIO and it keys them by name. This is how it routes a service request to the proper simulated device. Notice that the name is the tool name with a ".p" appended to the end for "provider". This convention is used automatically at the client end so you must use it here as well or else the request will not make it to the intended simulation object.

2.11.4.3 Receiving Commands from CCF

ISimulationIO defines a method, WriteOutputs(), that handles commands sent from the equipment control object in CCF and each tool simulation class must define this method. The commands arrive in the form of output values of endpoints. A simple implementation for a simulated valve is shown below.

```
void WriteOutputs(string ioProviderName, KeyValuePair<string, object>[] endpoints)
{
   foreach(KeyValuePair<string, object>[] endpoint in endpoints)
   {
      if( endpoint.Key == "ValveIsOpen" )
         this.oepSetOpen = (bool)endpoint.Value;
      if( endpoint.Key == "ValveIsClosed" )
         this.oepSetClosed = (bool)endpoint.Value;
   }
}
```

2.11.4.4 Sending Information Back to CCF

In order for the simulation object to send state changes, notifications, and alarms back to the equipment control object, it must write to the endpoint input values contained within the ScannedIOServer using the UpdateInputs() method. An example of simulation code doing this is shown below.

```
KeyValuePair<string, object>{} values = new KeyValuePair<string, object>[2];
values[0] = new KeyValuePair<string, object>("ioeIsOpen", this.ioeIsOpen);
values[1] = new KeyValuePair<string, object>("ioeIsClosed", this.ioeIsClosed);
SimulationIOServer.Instance.UpdateInputs("Valve.p", values);
```

These values are then picked up by the equipment control provider object in CCF.

Also as part of the ISimulationIO implementation, the simulation class must implement ISimulationIO.Initialize(string name). This handles a request made by the device provider object upon startup. It is a request for the tool to send back its initial state. An implementation of this method should include code very similar to what is shown above.

2.11.4.5 Equipment Control Modifications for Simulation

While the ScannedIOPointGroup object communicates with the simulated tool/IO device just like it is real hardware, it is still necessary to tell the ScannedIOPointGroup that the device on the other end is actually a simulation and not real hardware. This is so it can create the proper type of IIOProvider, in this case a SimulationIOProvider. To communicate this to the ScannedIOPointGroup object, make sure the following flag is set before starting the ScannedIOPointGroup.

```
Configuration.UsingRealHardware = false;
```

Also, the program starting the equipment control must include the necessary client WCF configuration in its app configuration file. An example of this is shown below. Change the address attribute as necessary.

```
  <system.serviceModel>
    <bindings>
      <netTcpBinding>
        <binding name="CcfDefaultBinding" transferMode="Buffered"
maxConnections="30"
            maxReceivedMessageSize="2147483647" receiveTimeout="Infinite">
          <readerQuotas maxArrayLength="2147483647"
maxStringContentLength="2147483647" />
          <security mode="None">
            <transport clientCredentialType="None" protectionLevel="None" />
            <message clientCredentialType="None" />
          </security>
        </binding>
      </netTcpBinding>
    </bindings>
    <client>
      <endpoint address="net.tcp://localhost:6415" binding="netTcpBinding"
        bindingConfiguration="CcfDefaultBinding"
contract="Cimetrix.CimControlFramework.IO.Scanned.Simulation.ISimulationIO"
```

```
        name="SimulationIOEndpoint" />
    </client>
</system.serviceModel>
```

2.12 Using CCF only for Supervisory Control

In some scenarios, CCF is used only for Supervisory Control. The equipment control and scheduling is handled by other software. In this role, CCF is primarily responsible for interfacing with equipment clients. The equipment clients could be some combination of Operator Interface, Factory Host, Interface A clients, and off-tool data analysis packages. You may wish to review the Architecture section.

2.12.1 Create Control Application

In this role, there is typically just a single CCF control application (see New CCF Control Application). You may wish to start with either the ToolSupervisor or SampleSupervisor control applications. Remove any CCF components that will not be used, such as the Equipment Control System, Scheduler, and/or OIServer.

Alternately, the required CCF packages could be incorporated into an existing application.

2.12.2 Create communication layer

There must be some type of communication between CCF and the rest of the equipment software. Choose a method appropriate to the equipment deployment. Some out-of-process options include WCF servers and straight TCP/IP. WCF is natively supported by some CCF packages. Other communication options usually require an additional, custom communication package in the CCF control application.

2.12.3 Load Port integration

Using the CCF load port classes is recommend because of the tightly coupled nature of load ports and Factory Automation interaction. Follow the instructions found in "Load Port" section. Instead of interacting directly with the load port hardware, the LoadPortProvider can interface with an intermediate object instead.

2.12.4 Adding Data Variables

Data variables are used to expose internal variables to the equipment clients. It is not necessary to expose all internal variables to the equipment clients—the equipment designer can choose which ones to expose.

For additional information, see "Creating and updating variables".

2.12.5 Events

Also known as collection events, events are used to expose important points of time to the equipment clients. Clients typically use the events to determine when to start and stop collection data variables. Some typical events would include Starting/Stopping of processing tasks. Again, the equipment designer can choose which events to expose.

For additional information, see "Creating and using events".

2.12.6 Alarms

Alarms are used to expose errors to the equipment clients and to solicit recovery actions. If an equipment uses alarms and is not using a CCF Operator interface, the Alarm server should be notified when recovery actions are selected.

For additional information, see “Creating and using execution alarms”, “Creating and using managed alarms”, and “Creating and using notification alarms”.

2.12.7 CIMPortal Model

Variables, events, and alarms are added to the CIMPortal model during creation. However, the hierarchy locations where they are added are not. Hierarchy nodes must be added manually. For additional information, see “Creating CIMPortal deployment packages”.

When variables, events, and alarms are added to the CIMPortal model, some assumptions about where it should be added are made. For example, a variable with the name “PM1.GateValve.isOpened” by default is added to the “Equipment\PM1\GateValve” hierarchy node with the name “isOpened”. If you want to change the way names are mapped to hierarchy nodes, override the `GetParentLocator` method of FactoryAutomationServer.

2.12.8 E90 Substrate Tracking

To enable E90 substrate tracking, use the following sections:

- “Adding a substrate location”
- “Moving a substrate”
- “Updating substrate object model”

2.12.9 E94 Control Jobs and E40 Process Jobs

To enable control job and process job management, use the following sections:

- “How to use job management”

2.12.10 E116 Equipment Performance Tracking

To enable E116 Equipment performance tracking, use the following sections:

- “How to use equipment performance tracking”

2.12.11 Notification Services

Notification services are used to pass command requests between different CCF packages when a tightly coupled relationship is not wanted. They are primarily used by the GUI to send commands to the control system. The parameters in the requests and responses are passed as sets of name/value pairs. This looser coupling allows new commands and parameters to be added without changing the communication layer, but prevents the compiler from detecting mismatches in what is expected by both sides.

A provider of a Notification service must register the service being provided. This can be done in two ways:

1. Register directly with the NotificationClient class

```
NotificationClient.RegisterService("ECS", "Initialize", "none", "initialize the
equipment control system and all subcomponents", Service_RequestPosted);
```

2. Creating a Service object.

```
Service service = new Service("Initialize", "ECS", "none", "initialize the
equipment control system and all subcomponents");
service.RequestPosted += new
EventHandler<RequestPostedEventArgs>(Service_RequestPosted);
```

When a service is requested, the registered event handler is called. An event handler can handle one or more different services requests. A typical pattern is to have one service event handler for a given instance.

```
/// <summary>
/// Handle notification service requests
/// </summary>
/// <param name="sender">Event sender</param>
/// <param name="e">Event data</param>
void Service_RequestPosted(object sender, RequestPostedEventArgs e)
{
    Response response = new Response(e.Request, ResponseCode.Performed, "");
    switch( e.Request.Service )
    {
        case "Initialize":
            // Do something here
            break;

        default:
            response.ResponseCode = ResponseCode.Rejected;
            response.ResultCode = "Unknown service";
            break;
    }

    // Send the response
    NotificationClient.PostResponse(response);
}
```

It is not necessary that the service be completed before the event handler returns. A service may be queued for later processing, in which case the Performed response should not be sent until the service was completed.

If a service is requested which has not been registered, a warning is logged.

2.12.12 Supporting GUI screens

Every GUI screen expects some functionality to be provided by the control system. Usually, this functionality is provided through publishing data variables and notification services.

Which variables are needed by a screen can be determined by source code inspection or by running the Operator Interface and changing to the screen in question. Data variables which are expected but missing are logged.

Required Notification services can be determined in much the same way—through inspection or trial. Notification services are typically invoked in response to a clicking on buttons.

3. Design Information

3.1 Alarms Package

3.1.1 Overview

The Alarms package is part of the core CIMControlFramework architecture. It is utilized by nearly every software component in the control system. The Alarms package provides a set of classes and interfaces that are used to manage alarms.

The Alarms package maintains and manages the collection of active and inactive alarms. Alarms are used by software components to publish warning and error conditions to the equipment operator. The operator can review the set of active alarms and select the desired recovery action. The desired recovery action is returned to the software component so the recovery action can be performed.

Each discrete equipment fault, warning condition, or error condition should have a unique alarm. Alarms should not be shared between software components. A software component which publishes an alarm is called the alarm owner.

Each alarm is associated with the following data:

- Name
- Id
- Code
- Message
- Description
- Recovery actions

3.1.1.1 Name

The name should be composed of an owner and an error condition to create a unique name. For example, a robot named "VTM.Robot" publishing a pick failure alarm could use an alarm named "VTM.Robot.PickFailed".

3.1.1.2 Id

The alarm id is a unique integer number used to identify the alarm to the GEM host. Because a GEM host may identify an alarm by its id alone, it's important that the alarm's id be consistent over time. When an alarm is first defined, the Alarms package will automatically assign it a unique id and create a persistent record of that assignment. Future alarm definition requests for that same alarm will result in the same id. The Alarms package assigns ids in id ranges, based on the alarm's name. For example, the two alarms named "VTM.Robot.PickFailed" and "VTM.Robot.PlaceFailed" will both be assigned ids in the id range assigned to the alarm owner "VTM.Robot".

3.1.1.3 Code

An enumerated byte code (SEMI E5 ALCD) categorizing the alarm.

3.1.1.4 Message

A brief text message (SEMI E5 ALTX) specifying the fault condition. The message is limited to 80 characters.

3.1.1.5 Description

A message describing the alarm, including possible root causes. If an alarm description is not provided when the alarm is defined, the Message will be used as a Description.

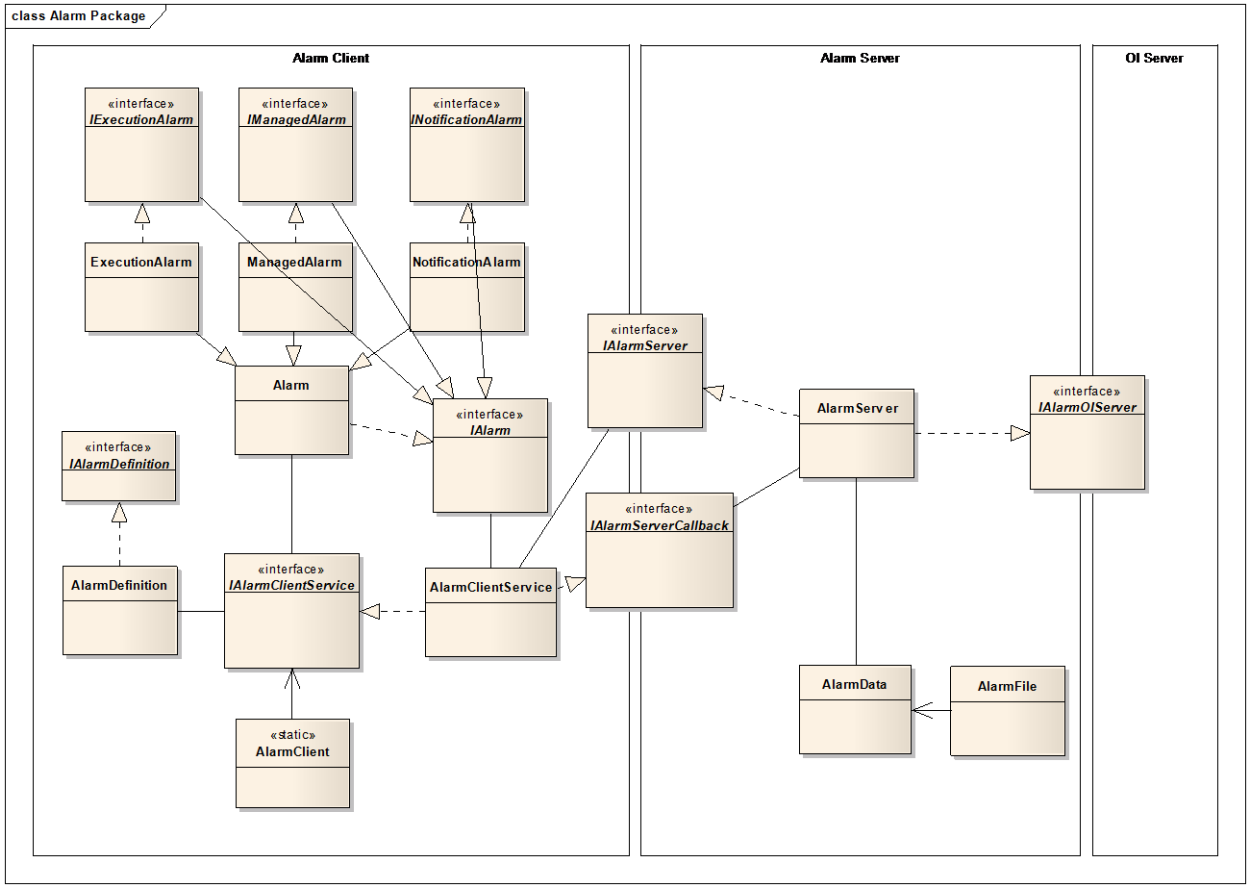
3.1.1.6 Recovery Actions

A set of recovery actions the operator may select for resolving the alarm fault condition. Each recovery action is limited to 20 characters and each alarm may have up to 5 recovery actions. Recovery actions can be enabled (selectable) or disabled (non-selectable). Only a single recovery alarm may be selected.

Alarms are activated and deactivated by their owners. Activating an alarm is called “posting”. Deactivating an alarm is called “clearing”. Recovery actions should be enabled only when the software is in a state that can execute recovery actions.

The Alarms package is implemented in Cimetrix.Alarms.dll.

3.1.2 Class Diagram



3.1.3 Class Descriptions

The Alarms package interface and class properties and methods are detailed in the reference API documentation.

Alarm Client classes are used by the alarm owners. The Alarm Client class instances will be located in the same process as alarm owners.

3.1.3.1 IExecutionAlarm

An interface for execution alarms. Execution alarms provide the alarm owner with explicit control of the alarm states. Posting and clearing of the alarm active state as well as enabling and disabling of recovery actions are controlled directly by the alarm owner. Execution alarms should be used for all alarms which have recovery actions other than "Acknowledge".

3.1.3.2 ExecutionAlarm

An execution alarm class which implements the default CCF execution alarm behavior. If needed, a project class may be derived from ExecutionAlarm to override the default behavior. Alternately, a new class implementing the IExecutionAlarm interface may be created.

To allow CCF software packages to use the project execution alarm class, a new alarm client service must be created. The implementation of the CreateExecutionAlarm method should create an instance of the new project execution alarm class instead of an ExecutionAlarm.

3.1.3.3 IManagedAlarm

An interface for managed alarms. Managed alarms are used to publish a fault condition of finite (non-instantaneous) duration. The managed alarm owner is limited to the publishing of a Boolean indicating the fault condition state (active, inactive). They have a single recovery action of "Acknowledge".

Managed alarms are posted when the fault condition is active. The "Acknowledge" recovery action is disabled while the fault condition state is active. When the fault condition becomes inactive, the managed alarm is either automatically cleared, or only cleared when the operator selects the "Acknowledge" recovery action. This behavior is based on configuration.

An example of a managed alarm would be a software component losing communications with a hardware device, assuming the software automatically will reconnect when hardware device comes back online. The software component would set the fault condition to active while the hardware device is offline, and back to inactive once communications had been restored.

3.1.3.4 ManagedAlarm

A managed alarm class which implements the default CCF managed alarm behavior. If needed, a project class may be derived from ManagedAlarm to override the default behavior. Alternately, a new class implementing the IManagedAlarm interface may be created.

To allow CCF software packages to use the project managed alarm class, a new alarm client service must be created. The implementation of the CreateManagedAlarm method should create an instance of the new project execution alarm class instead of a ManagedAlarm.

3.1.3.5 INotificationAlarm

An interface for notification alarms. Notification alarms are used to publish an instantaneous fault condition. The notification alarm owner is limited to a single "Post" method. They have a single recovery action of "Acknowledge".

Notification alarms are posted when the alarm owner calls the "Post" method. The "Acknowledge" recovery action is always enabled. The alarm is cleared when the operator selects the "Acknowledge" recovery action or automatically after a configurable amount of time.

An example of a notification alarm would be an invalid command. The invalid command is detected and discarded with no recovery needed.

3.1.3.6 NotificationAlarm

A notification alarm class which implements the default CCF managed alarm behavior. If needed, a project class may be derived from NotificationAlarm to override the default behavior. Alternately, a new class implementing the INotificationAlarm interface may be created.

To allow CCF software packages to use the project notification alarm class, a new alarm client service must be created. The implementation of the CreateNotificationAlarm method should create an instance of the new project execution alarm class instead of a NotificationAlarm.

3.1.3.7 IAlarm

A base interface for methods common to all alarm types. This interface is not generally used by alarm owners and cannot be used for posting and clearing alarms.

3.1.3.8 Alarm

A base class which implements functionality common to all alarm types.

3.1.3.9 IAlarmDefinition

The interface that defines the content of an alarm definition. This includes the alarm name, code, message, description, and recovery actions.

3.1.3.10 AlarmDefinition

The default implementation of the IAlarmDefinition interface. An alarm definition is used by alarm owners to define a new alarm at CCF startup. The alarm owner creates the new alarm by passing this definition in a call to the IAlarmClientService's CreateExecutionAlarm, CreateManagedAlarm, or CreateNotificationAlarm method.

3.1.3.11 IAlarmClientService

An interface for the alarm client services. Alarm client services are responsible for abstracting the communications between the alarm owners (and their alarms) and the alarm server. The alarm client service is used to register alarms with the alarm server, retrieve registered alarms, and create alarm instances. There is an alarm client service in each application using CCF alarms.

3.1.3.12 AlarmClientService

An alarm client service class which implements the default CCF alarm client service behavior. If needed, a project class may be derived from AlarmClientService to override the default behavior. Alternately, a new class implementing the IAlarmClientService interface may be created.

In either case, the application should instantiate and store the project alarm client service in the AlarmClient class before any software components dependent on alarm client service are created.

The alarm client service is also responsible for re-registering alarms when the connection to the alarm server is restored (after being lost). At this time, all posted alarms should be re-posted.

The AlarmClientService can be used to communicate with an in-process alarm server or a WCF alarm server host.

3.1.3.13 AlarmClient

A static service locator for the alarm client service. Software components which need to use the alarm client service should use the AlarmClient class to find the correct IAlarmClientService interface.

3.1.3.14 IAlarmServerCallback

The interface used by AlarmServer to communicate with an AlarmClient. Used for information and services related to alarm owners.

3.1.3.15 IAlarmServer

The interface used by AlarmClient to communicate with the AlarmServer. Used for information and services related to alarm owners. There is a single alarm server per equipment. All alarm clients use the same alarm server. The alarm server is responsible for maintaining a list of all active alarms.

If the connection between the alarm client and the alarm server is lost, all posted alarms from that alarm client are cleared.

The alarm server is responsible for reading stored alarm definitions. As alarm owners start up, they register their alarms with the Alarms package. Each alarm may only be registered by one owner and if an alarm has already been registered, a registration error will occur.

3.1.3.16 AlarmServer

An alarm server class which implements the default CCF alarm server behavior. If needed, a project class may be derived from AlarmServer to override the default behavior. Alternately, a new class implementing the IAlarmServer and IAlarmOIServer interfaces may be created. In either case, the new alarm server should be instantiated by the server host application.

3.1.3.17 AlarmData

Class used by the AlarmServer to hold the data associated with an alarm.

3.1.3.18 IAlarmOIServer

An interface used by OIServer to communicate with the Alarm Server. Used for information and services related to the GUI.

3.1.4 Samples

3.1.4.1 Creating and using execution alarms

At CCF startup, the data models for both CIMPortal and CIMConnect will be automatically updated with the alarms defined by the software components during their creation.

To add an execution alarm, call the Alarm Client's RegisterAlarms method from the constructor and supply one or more IAlarm interfaces from Alarm Client's CreateExecutionAlarm. The following arguments are used in creating each new execution alarm:

- Name – A name for the new alarm. This name must be unique across CCF and usually includes a name, for example: "EfemRobot.PickFailed". The CxPath.Combine method, which uses the '.' character as a delimiter, is recommended for combining the elements of alarms names into a single string. The alarm name is used to determine where in the CIMPortal hierarchy model the parameter will be added. For example, a parameter named "EfemRobot.PickFailed" would be added to the node whose CIMPortal locator would be Equipment\EfemRobot.
- AlarmCode – The GEM code for the alarm. Most execution alarms should be AlarmCode.IrrecoverableError because the software cannot recover without intervention from the operator.
- Message – A short (80 characters or less) message description of the alarm.
- Description – A human-readable description of the alarm.
- RecoveryActions – An array of valid recovery actions for the alarm. The operator will chose a recovery action to indicate how the software is to proceed. There can be no more than 5 recovery actions and each recovery action is limited to 20 characters.

The following code sample shows how to create a new execution alarm and how to register the new alarm using the Alarm Client's RegisterAlarms method.

```
// Create the local alarm object that will be used to control the alarm
IExecutionAlarm pickFailedAlarm = AlarmClient.Service.CreateExecutionAlarm(
    "EfemRobot.PickFailed",
    AlarmCode.IrrecoverableError,
    "Efem robot pick command failed",
```

```
    " More detailed alarm description goes here ",
    new string[] { "Retry", "Fail" });

// Register the alarm with the alarm server
AlarmClient.Service.RegisterAlarms(new IAlarm[] { pickFailedAlarm });
```

Note that the RegisterAlarms method must be called in the constructor. If the class derives from the ActiveObject class, the call must be made prior to the ActiveObject.Start() method.

The CreateExecutionAlarm method returns an IExecutionAlarm instance that is retained for controlling the alarm, as shown in the code samples below. The execution alarm is then registered with the alarm server, using the RegisterAlarms method, which makes the alarm ready for use and updates the CIMPortal and CIMConnect models.

*Note: The processing of the CreateExecutionAlarm method takes place locally. The RegisterAlarms method, however, may involve communication with a remote alarm server so it may be advantageous to consolidate the IAlarm instances that result from calling Create*Alarm and provide all of them to a single call to RegisterAlarms.*

Alarms can be created and registered at any time, prior to their use. However, if the alarm is to be available through CIMPortal (EDA Interface) and CIMConnect (GEM Interface), the alarm must be registered before the application hosting the Factory Automation Server component calls the DeployDataModels method. The DeployDataModels method insures that the alarm definition is included in the CIMPortal and CIMConnect data models.

When the error condition corresponding to this alarm occurs (in this case, a failed EFEM robot pick), the alarm owner posts the alarm and waits for the operator to select a recovery action:

```
// Something bad happens and the robot fails during pick command
pickFailedAlarm.SetRecoveryActionsEnabled(true);
pickFailedAlarm.Post();

// Wait for recovery action
pickFailedAlarm.RecoveryActionEventWaitHandle.WaitOne();
```

Once the recovery action has been selected, the alarm should be cleared. The selected recovery action is then carried out:

```
pickFailedAlarm.Clear(); // If retry fails, alarm will be re-posted

switch (pickFailedAlarm.SelectedRecoveryAction)
{
    case "Retry":
        // Retry pick
        break;
    case "Fail":
        // Fail the pick command and transition scheduler to Manual mode
        break;
    default:
        // Unknown recovery action
        break;
```

```
}
```

3.1.4.2 Creating and using managed alarms

At CCF startup, the data models for both CIMPortal and CIMConnect will be automatically updated with the alarms defined by the software components during their creation.

To add a managed alarm, call the Alarm Client's RegisterAlarms method from the constructor and supply one or more IAlarm interfaces from Alarm Client's CreateManagedAlarm. The following arguments are used in creating each new managed alarm:

- Name – A name for the new alarm. This name must be unique across CCF and usually includes a name, for example: "EfemRobot.PickFailed". The CxPath.Combine method, which uses the '.' character as a delimiter, is recommended for combining the elements of alarms names into a single string. The alarm name is used to determine where in the CIMPortal hierarchy model the parameter will be added. For example, a parameter named "EfemRobot.PickFailed" would be added to the node whose CIMPortal locator would be Equipment\EfemRobot.
- AlarmCode – The GEM code for the alarm.
- Message – A short (80 characters or less) message description of the alarm.
- Description – A human-readable description of the alarm.

The following code sample shows how to create a new managed alarm and how to register the new alarm using the Alarm Client's RegisterAlarms method.

```
// Create the local alarm object that will be used to control the alarm
IManagedAlarm communicationAlarm = AlarmClient.Service.CreateManagedAlarm(
    "DP2.CommunicationFailure",
    AlarmCode.EquipmentStatusWarning,
    "Communication failure with the pump",
    "More detailed alarm description goes here");

// Register the alarm with the alarm server
AlarmClient.Service.RegisterAlarms(new IAlarm[] { communicationAlarm });
```

The CreateManagedAlarm method returns an IManagedAlarm instance that is retained for controlling the alarm, as shown in the code sample below. The managed alarm is then registered with the alarm server, using the RegisterAlarms method, which makes the alarm ready for use and updates the CIMPortal and CIMConnect models.

*Note: The processing of the CreateManagedAlarm method take place locally. The RegisterAlarms method, however, may involve communication with a remote alarm server so it may be advantageous to consolidate the IAlarm instances that result from calling Create*Alarm and provide all of them to a single call to RegisterAlarms.*

Alarms can be created and registered at any time, prior to their use. However, if the alarm is to be available through CIMPortal (EDA Interface) and CIMConnect (GEM Interface), the alarm must be registered before the application hosting the Factory Automation Server component calls the DeployDataModels method. The DeployDataModels method insures that the alarm definition is included in the CIMPortal and CIMConnect data models.

To use a managed alarm, the alarm owner updates the fault condition of the communication alarm, either periodically or on-change:

```
// Update fault condition of communication alarm
communicationAlarm.FaultCondition = (initialized && serialPort == null);
```

It is usually easier to update the fault condition periodically. The alarm owner does not need to worry about posting or clearing the alarm directly.

3.1.4.3 Creating and using notification alarms

At CCF startup, the data models for both CIMPortal and CIMConnect will be automatically updated with the alarms defined by the software components during their creation.

To add a notification alarm, call the Alarm Client's RegisterAlarms method from the constructor and supply one or more IAlarm interfaces from Alarm Client's CreateNotificationAlarm. The following arguments are used in creating each new notification alarm:

- Name – A name for the new alarm. This name must be unique across CCF and usually includes a name, for example: "EfemRobot.PickFailed". The CxPath.Combine method, which uses the '.' character as a delimiter, is recommended for combining the elements of alarms names into a single string. The alarm name is used to determine where in the CIMPortal hierarchy model the parameter will be added. For example, a parameter named "EfemRobot.PickFailed" would be added to the node whose CIMPortal locator would be Equipment\EfemRobot.
- AlarmCode – The GEM code for the alarm.
- Message – A short (80 characters or less) message description of the alarm.
- Description – A human-readable description of the alarm.

The following code sample shows how to create a new notification alarm and how to register the new alarm using the Alarm Client's RegisterAlarms method.

```
// Create the local alarm object that will be used to control the alarm
INotificationAlarm modelDeploymentFailedAlarm =
AlarmClient.Service.CreateNotificationAlarm(
"FactoryAutomation.DynamicModelDeploymentFailure",
AlarmCode.DataIntegrity,
"Dynamic model deployment failure",
"More detailed alarm description goes here");

// Register the alarm with the alarm server
AlarmClient.Service.RegisterAlarms(new IAlarm[] { modelDeploymentFailedAlarm
});
```

The CreateNotificationAlarm method returns an INotificationAlarm instance that is retained for controlling the alarm, as shown in the code sample below. The notification alarm is then registered with the alarm server, using the RegisterAlarms method, which makes the alarm ready for use and updates the CIMPortal and CIMConnect models.

*Note: The processing of the CreateNotificationAlarm method take place locally. The RegisterAlarms method, however, may involve communication with a remote alarm server so it may be advantageous to consolidate the IAlarm instances that result from calling Create*Alarm and provide all of them to a single call to RegisterAlarms.*

Alarms can be created and registered at any time, prior to their use. However, if the alarm is to be available through CIMPortal (EDA Interface) and CIMConnect (GEM Interface), the alarm must be registered before the application hosting the Factory Automation Server component calls the

DeployDataModels method. The DeployDataModels method insures that the alarm definition is included in the CIMPortal and CIMConnect data models.

When the error condition corresponding to this alarm occurs (in this case, a model deployment failure) the alarm owner posts the alarm. The alarm owner does not need to worry about clearing the alarm directly:

```
// Model failed to deploy properly
modelDeploymentFailedAlarm.Post("Error deploying model");
```

3.1.4.4 How to create a custom AlarmServer

In the project specific code, create a new alarm server class (called MyAlarmServer in this example). This project class may be derived from AlarmServer, or may implement the IAlarmServer and IAlarmOIServer interfaces. Implement the desired functionality in the new class. The application hosting the alarm server should be modified to create the new project alarm server.

Do this by changing:

```
alarmServer = new AlarmServer("Alarms.xml", "AlarmNotes.xml");
```

To:

```
alarmServer = new MyAlarmServer("Alarms.xml", "AlarmNotes.xml");
```

3.1.4.5 How to create a custom AlarmClientService

In the project specific code, create a new alarm client service class (called MyAlarmClientService in this example). This project class may be derived from AlarmClientService, or may implement the IAlarmClientService interface. Implement the desired functionality in the new class.

An example deriving from AlarmClientService would be:

```
class MyAlarmClientService : AlarmClientService
{
    public override IExecutionAlarm CreateExecutionAlarm(string name)
    {
        return new MyExecutionAlarm(name);
    }
}
```

Each project application that uses the alarm client (including the application hosting the alarm server) should set the Service property on the AlarmClient class before starting to use the alarm client service.

```
AlarmClient.Service = new MyAlarmClientService();
```

3.1.4.6 How to create a custom ExecutionAlarm (or ManagedAlarm or NotificationAlarm)

In the project specific code, create a new execution alarm class (called MyExecutionAlarm in this example). This project class may be derived from ExecutionAlarm, or may implement the IExecutionAlarm interface. Implement the desired functionality in the new class.

An example deriving from ExecutionAlarm would be:

```
class MyExecutionAlarm : ExecutionAlarm
{
    public MyExecutionAlarm(string name)
        : base (name)
    {
        Source = CxPath.Combine("MyExecutionAlarm", name);
    }

    public override void Post()
    {
        Log.WriteIfEnabled(LogCategory.Debug, Source, "Example overriding Post");
        base.Post();
    }
}
```

The alarm client service needs to be modified to create the new execution alarm class in the "CreateExecutionAlarm" method. See "How to create a custom AlarmClientService" for details.

3.1.4.7 How to create a custom alarm categories

For some projects, it may be desirable to separate alarms into different categories. One way custom alarm categories can be done in CCF is by creating a custom alarm server. Each time an alarm is posted, the custom alarm server can determine the category, salience type, and control if the alarm is sent to the GEM host or not.

By default, the CCF AlarmServer uses the E5 alarm code as the category, sets the salience type to Alarm, and always sends the alarm to the GEM host.

An example of how to separate alarms into two categories ("Alarms" and "Warnings") follows:

```
/// <summary>
/// Class for customizing alarm post categories, saliences, and sending to GEM
host.
/// </summary>
class MyAlarmServer : AlarmServer
{
    public MyAlarmServer(string alarmFileName, string alarmNoteFileName)
        : base(alarmFileName, alarmNoteFileName)
    {
    }

    /// <summary>
    /// Gets the category of the alarm post.  Categories can be any set of strings.
    /// Categories should not contain spaces or control characters.
    /// </summary>
    /// <param name="alarm">Alarm being posted</param>
    /// <returns>The category name for this alarm posting</returns>
    protected override string GetCategory(AlarmData alarm)
    {
        if( alarm.Code == AlarmCode.EquipmentStatusWarning ||
            alarm.Code == AlarmCode.ParameterControlWarning )
        {
            return "Warning";
        }
```

```
            return "Alarm";
        }

        /// <summary>
        /// Gets the salience of the alarm post.  The salience will be used by the
operator interface
        /// to determine the salience around the alarm screen set navigation button.
        /// </summary>
        /// <param name="alarm">Alarm being posted</param>
        /// <returns>The salience for this alarm posting</returns>
        protected override SalienceType GetSalience(AlarmData alarm)
        {
            if( alarm.Category == "Warning" )
                return SalienceType.Caution;
            return SalienceType.Alarm;
        }

        /// <summary>
        /// Called when the alarm should be sent to the GEM host
        /// </summary>
        /// <param name="alarm">Alarm being posted</param>
        /// <returns>True if alarm should be sent to GEM host, otherwise
false</returns>
        protected override bool GetSendToGemHost(AlarmData alarm)
        {
            if( alarm.Category == "Warning" )
                return false;
            return true;
        }
}
```

See “How to create a custom AlarmServer” for additional details on how to create custom alarm servers.

3.1.4.8 How to use WCF to connect AlarmServer and AlarmClientService

The CCF AlarmClientService does not need to be in the same application as the CCF AlarmServer. They can communication with each other using WCF (Windows Communication Foundation).

To configure the CCF AlarmClientService to use WCF to communicate with the AlarmServer, information must be added to the configuration file of the application with the AlarmClientService. In a Visual Studio project, this file is usually named “App.config”. After building the project, the “App.config” file is copied to the output folder and renamed “xxxx.exe.config”, where xxxx is the application name. To use the TCP/IP transport layer, the client application should have an entry similar to the following, where localhost is the name of the computer hosting the AlarmServer and 6401 is the port number on which the server is listening:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <bindings>
      <netTcpBinding>
        <binding name="CcfDefaultBinding" transferMode="Buffered"
maxConnections="30"
          maxReceivedMessageSize="2147483647">
          <readerQuotas maxArrayLength="2147483647" />
```

```
          <security mode="None">
            <transport clientCredentialType="None" protectionLevel="None" />
            <message clientCredentialType="None" />
          </security>
        </binding>
      </netTcpBinding>
    </bindings>
    <client>
      <endpoint address="net.tcp://localhost:6401" binding="netTcpBinding"
          bindingConfiguration="CcfDefaultBinding"
contract="Cimetrix.CimControlFramework.Alarms.IAlarmServer"
          name="AlarmEndpoint" />
    </client>
  </system.serviceModel>
</configuration>
```

To configure the CCF AlarmServer to use WCF to communicate with the alarm client services, information must be added to the configuration file of the application host the AlarmServer. To use the TCP/IP transport layer, the server application should have an entry similar to the following, where localhost is the name of the computer hosting the AlarmServer and 6401 is the port number on which the server should listen:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <bindings>
      <netTcpBinding>
        <binding name="CcfDefaultBinding" transferMode="Buffered"
maxConnections="30"
          maxReceivedMessageSize="2147483647">
          <readerQuotas maxArrayLength="2147483647" />
          <security mode="None">
            <transport clientCredentialType="None" protectionLevel="None" />
            <message clientCredentialType="None" />
          </security>
        </binding>
      </netTcpBinding>
    </bindings>
    <services>
      <service name="Cimetrix.CimControlFramework.Alarms.AlarmServer">
        <endpoint address="net.tcp://localhost:6401" binding="netTcpBinding"
            bindingConfiguration="CcfDefaultBinding"
contract="Cimetrix.CimControlFramework.Alarms.IAlarmServer" />
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

Note that the bindings on the client and server sides must match. If a custom alarm server is being used, the service name attribute “Cimetrix.CimControlFramework.Alarms.AlarmServer” must be changed to match the namespace and class name of custom alarm server.

In addition, the server application must call the StartServiceHost method on AlarmServer.

3.2 Factory Automation Package

3.2.1 Overview

The Factory Automation (FA) package is part of the core CIMControlFramework architecture. The Factory Automation package provides a set of classes and interfaces that are used to manage their alarms in the system.

The primary purpose of the Factory Automation package is to provide CCF access to functionality specified in SEMI standards. To achieve this purpose, the FA package provides support for data publication, substrate tracking, job management, and equipment performance.

The Factory Automation package consists of two assemblies.

FactoryAutomationClient service—the primary interface between CCF and the Factory Automation package.

FactoryAutomation server—the server side of FA package. Normally only used by the FA client server, the FA server can be modified through extension and substitutability to modify/replace default CCF behavior with project specific behavior. Some packages also access FA server load port interfaces directly. Directly accessing FA server functionality is discouraged and is likely to be removed in future versions.

3.2.1.1 Data Publication

Data publication is the ability for a software component in the CCF solution to expose a variable value to other software components. The FA package can expose these variable values to CIMConnect, which further exposes them to a GEM host connection. The FA package also exposes these variable values to CIMPortal. CIMPortal acts as a data router, sending the data on to Interface A (through CIMWeb), a historical data base (through CIMStore), and even back to CCF operator interfaces (through OIServer).

Each variable value must have a unique name. To help prevent name conflicts, it is recommended that each software component prepend the software component name to its variable names, with a period delimiter. For example, if a load lock named “LL1” was publishing a variable named “Pressure”, the full variable name would be “LL1.Pressure”.

Data publication variable values can either be updated periodically or just when they change. Internally, the data publication software keeps track of the previous values and only sends updates when the variable values change.

Data publishing is designed so that data variable values in a set are always updated together. This is called an “atomic operation”. “Atomic” in this sense means indivisible. In practical terms, this means that if a query is done to get the variable values, either all or none of the values will have been updated and not half of the new values and half of the old ones.

Data variables (also known as parameters) can be defined statically, i.e. before CCF is started, or dynamically when CCF starts. The procedure for the static definition of parameters, as well as events, exceptions, and alarms, can be found in Section **Error! Reference source not found.Error! Reference urce not found.**. The procedure for defining data parameters dynamically is provided in Section a.

3.2.1.2 Substrate Tracking

Substrate Tracking is used to keep track of substrates as they are moved through the equipment from one substrate location to the next. Substrate Tracking keeps CIM300, CIMConnect, and the CCF material map updated. When substrates are moved, the FA package must be notified so it can update all interested parties.

Substrate transport and processing states are also tracked.

3.2.1.3 Job Management

Job management tracks all the created jobs and their states. When a job is ready to be executed, it is sent to the equipment scheduler. The scheduler determines and coordinates the series of commands that is needed to complete the job.

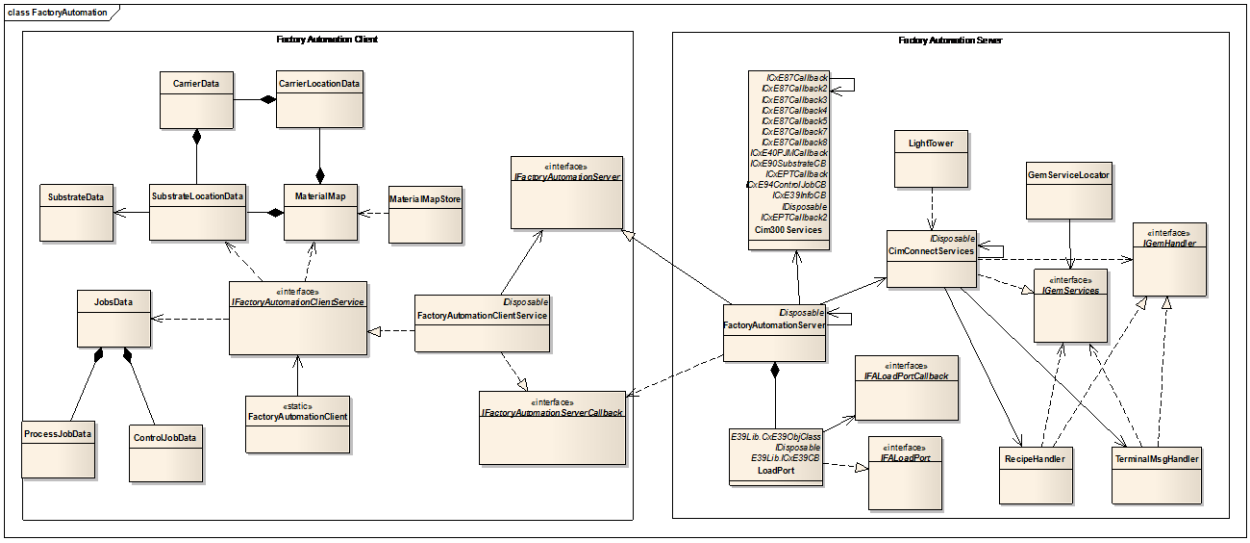
3.2.1.4 Equipment Performance Tracking

Equipment performance tracking monitors the current states of all equipment modules in the system. This information is used to collect information to comply with the E116 standard. Each time an equipment module starts or stops doing a task, the FA package should be notified.

The Factory Automation package is implemented in Cimetrix.FactoryAutomationClient.dll (client side) and Cimetrix.FactoryAutomation.dll (server side).

For the FA package to work correctly, both CIMConnect and CIMPortal must also be configured correctly.

3.2.2 Class Diagram



3.2.3 Class Descriptions

The Factory Automation package interface and class properties and methods are detailed in the reference API documentation.

Accessing the functionality of the FA package is primarily done through the IFactoryAuomationClientService interface.

3.2.3.1 IFactoryAutomationClientService

This is an interface for the factory automation client services. Factory Automation client services are responsible for abstracting the communications to the Factory Automation server. There is a client service in each application using Factory Automation package.

3.2.3.2 FactoryAutomationClientService

This is a Factory Automation client service class which implements the default CCF factory automation client service behavior. If needed, a project class may be derived from FactoryAutomationClientService to override the default behavior. Alternately, a new class implementing the IFactoryAutomationClientService interface may be created.

In either case, the application should instantiated and store the project alarm client service in the FactoryAutomationClientService class before any software components dependent on factory automation client service are created.

The FactoryAutomationClientService can be used to communicate with an in-process factory automation server or a WCF factory automation server host.

3.2.3.3 FactoryAutomationClient

This is a static service locator for the factory automation client service. Software components which need to use the factory automation client service should use the FactoryAutomationClient class to find the correct IFactoryAutomationClientService interface.

3.2.3.4 ProcessJobData

A data class used by the Factory Automation package to contain data related to a specific process job.

3.2.3.5 Control Job Data

A data class used by the Factory Automation package to contain data related to a specific control job.

3.2.3.6 JobsData

A data class used by the Factory Automation package to contain data related to all process and control jobs that exist in the equipment.

3.2.3.7 SubstrateData

A data class used by the Factory Automation package to contain data related to a specific substrate.

3.2.3.8 SubstrateLocationData

A data class used by the Factory Automation package to contain data related to a specific location in the equipment that is capable of holding a substrate.

3.2.3.9 CarrierData

A data class used by the Factory Automation package to contain data related to a specific carrier in the equipment.

3.2.3.10 CarrierLocationData

A data class used by the Factory Automation package to contain data related to a specific location in the equipment that is capable of holding a carrier.

3.2.3.11 MaterialMap

A data class used by the Factory Automation package to contain data related to all substrates, substrate locations, carriers, and carrier locations that exist in the equipment.

3.2.3.12 MaterialMapStore

A class responsible for providing persistence for the current material map. The material map store provide functionality for both reading and writing the material map.

3.2.3.13 IFactoryAutomationServerCallback

The interface used by FactoryAutomationServer to communicate with a FactoryAutomationClient.

3.2.3.14 IFactoryAutomationServer

The interface used by FactoryAutomationClient to communicate with the FactoryAutomationServer. There is a single factory automation server per equipment. All clients use the same factory automation server.

The factory automation server is responsible for data publication, substrate tracking, job management, and equipment performance tracking.

3.2.3.15 FactoryAutomationServer

A factory automation server class which implements the default CCF behavior. If needed, a project class may be derived from FactoryAutomationServer to override the default behavior. It is not yet possible to use substitution by creating a new class implementing the IFactoryAutomationServer interface. The new factory automation server should be instantiated by the server host application.

3.2.3.16 CimConnectServices

A singleton class used to communicate with CIMConnect. It implements IGemServices.

3.2.3.17 IGemServices

The interface used to communicate with the GEM services provider. There is a single GEM services provider per equipment. All clients use the same factory automation server.

3.2.3.18 IGemHandler

The interface implemented by users of the GEM services provider to receive callbacks when messages have been received.

3.2.3.19 GemServiceLocator

This is a static service locator for the GEM services. Software components which need to use GEM services should use the GemServiceLocator class to find the correct IGemServices interface.

3.2.3.20 RecipeHandler

RecipeHandler uses IGemServices/IGemHandler to exchange unformatted recipes with the factory host. It uses Notifications to interact with the GUI to notify the user and receive user requests for recipe transfers.

3.2.3.21 TerminalMsgHandler

TerminalMsgHandler uses IGemServices/IGemHandler to exchange terminal messages with the factory host. It uses Notifications to interact with the GUI to notify the user and receive user requests for chatting using terminal messages services.

3.2.3.22 Cim300Services

A singleton class used to communicate with CIM300.

3.2.3.23 IFALoadPort

An interface used by CCF to communicate to a factory automation loadport. Use of the IFALoadPort interface from non-FA software modules is discouraged.

3.2.3.24 IFALoadPortCallback

An interface used by a factory automation loadport to communicate with other CCF software modules. Typically implemented by load port providers (drivers).

3.2.3.25 LoadPort

A class used by the factory automation server to represent a load port.

3.2.3.26 LightTower

A class used by the factory automation server to represent a light tower. The FA LightTower class uses CIMConnect to determine the correct state of each light. These light states are then sent to the light tower providers (drivers) to set the actual hardware light states.

3.2.4 Notification Services

In addition to the notification services listed below, the FA package can translate GEM remote commands into CCF notification service calls. This functionality is contained in the CimConnectServices::OnRemoteCommandCalled method.

3.2.4.1 Provided Notification Services

These notification services are provided by the default Factory Automation packages and may be requested by other software components.

Provider	Service Identifier	Description
FA	CreateJob	Local job creation.
FA	JobCommand	Local job commands (stop, resume, pause, abort).
FA	PJCommand	Process job (PJ) commands (stop, resume, pause, abort).
FA	CJCommand	Control job (CJ) commands (stop, resume, pause, abort).
FA	CreatePJ	Create a process job.
FA	CreateCJ	Create a control job.
FA	Bind	Associate a carrierId to a load port.
FA	CancelBind	Disassociate a carrierId from a load port.
FA	CancelCarrier	Cancel a carrier at a load port, make it ready to unload.
FA	ReCreateCarrier	Perform an E87 CarrierReCreate.
FA	ProceedWithCarrier	Proceed with the carrier at the specified port.
FA	ReservePort	Change the reservation status of a port.
FA	StartWaferCycling	Continuously cycle selected wafers from a FOUP.
FA	StopWaferCycling	Stop cycling selected wafers from a FOUP.
FA	PauseWaferCycling	Pause cycling selected wafers from a FOUP.
FA	ResumeWaferCycling	Resume cycling selected wafers from a FOUP.
FA	StartWaferCyclingWithRouteRecipe	Continuously cycle selected wafers from a FOUP using a route recipe.

FA	RequestFormattedRecipe	Request a formatted recipe from the host.
FA	SendFormattedRecipe	Send a formatted recipe to the host.
FA	RequestRecipe	Request a recipe from the host.
FA	SendRecipe	Send a recipe to the host.
FA	SetGEMControlState	Set the GEM control state.
FA	EnableGEMCommunicationState	Enable/Disable GEM communication.
FA	EnableSpooling	Enable/Disable GEM spooling.
FA	SpoolingOverwrite	Enable/Disable GEM spooling overwrite mode.
FA	SendTerminalMsg	Send a terminal message.
FA	TerminalMsgPosted	POIServer posting status.
FA	AcknowledgeTerminalMsg	Acknowledge a terminal message.
LightTower	ConfigLightTower	Configure Light Tower.
LPx	Clamp	Clamp the FOUP.
LPx	Unclamp	Unclamp the FOUP.
LPx	Dock	Dock the FOUP.
LPx	Undock	Undock the FOUP.
LPx	Load	Load the FOUP.
LPx	Open	Open the FOUP.
LPx	Close	Close the FOUP.
LPx	Unload	Unload the FOUP.
LPx	ReadCarrierId	Read the carrier id.
LPx	WriteCarrierId	Write the carrier id.
LPx	Map	Map the carrier.
LPx	SetManualMode	Set accessing mode to manual.
LPx	SetAutoMode	Set accessing mode to auto.
LPx	PutInService	Put the load port in service.
LPx	PutOutOfService	Put the load port out of service.
LPx	ReCreateCarrier	Perform an E87 CarrierReCreate.
LPx	E84Reset	Reset the E84 signals according to the accessing mode.

LPx	LoadButton	Load port Load button.
LPx	UnloadButton	Load port Unload button.
ThroughputMonitor	ResetThroughput	Reset the throughput calculator.
ThroughputMonitor	LogThroughput	Log the current throughput with the user provided message.

3.2.4.2 Requested Notification Services

These notifications services are requested by the default Factory Automation package and must be provided by another software component:

Provider	Service Identifier
OIServer	GUIUpdate
OIServer	TerminalMsgError
OIServer	TerminalMsgRcvd
OIServer	TerminalMsgAck
OIServer	VerificationDone
OIServer	VerifyCID
OIServer	VerifySM

3.2.5 Configuration Parameters

The following configuration parameters are used by the default Factory Automation package.

3.2.5.1 Branch: HostInterface.x

These configuration parameters are needed by each host interface connection.

Name	Type	Description
ActiveMode	BoolType	If TRUE, then the equipment will actively try to connect to the host using ActiveModeHostIPAddress and ActiveModeHostPortNumber
ActiveModeHostIPAddress	StringType	The IP address of the GEM host
ActiveModeHostPortNumber	IntType	The TCP port for the GEM Host
DeviceId	IntType	The device-id identifies the equipment and will be assigned by the factory. Common device-ids are 0 and 32767.
LinktestTimerInMsec	IntType	HSMS LINKTEST.REQ heartbeat message timer
PassiveModeIPAddress	StringType	The IP address of the NIC to use for GEM Host connections.
PassiveModePortNumber	IntType	The TCP port number on which to listen for host

		connections.
T3TimeoutInMsec	IntType	Transaction Timer
T5TimeoutInMsec	IntType	Connect separation timeout
T6TimeoutInMsec	IntType	Control transaction timeout
T7TimeoutInMsec	IntType	NOT SELECTED timeout
T8TimeoutInMsec	IntType	Network inter-character timeout

3.2.5.2 Branch: FactoryAutomation

Name	Type	Description
MDLNValue	StringType	GEM Equipment Model Type
ThroughputIntervalInSeconds	IntType	The interval of time in seconds between throughput updates

3.2.5.3 Branch: FactoryAutomation.DataPublication

Name	Type	Description
DataPublicationInterval	IntType	Timer interval (in milliseconds) for Factory Automation clients to send changed data variables to the Factory Automation server.
GemSubsampleInterval	IntType	Timer interval (in milliseconds) to for Factory Automation server to send data to CIMConnect
MaxDataQueueSize	IntType	Maximum size of the gem subsample data queue before entries are thrown out

3.2.5.4 Branch: FactoryAutomation.LightTowerConditions

The FactoryAutomation.LightTowerConditions branch does not have a fixed set of configuration parameters. The parameters can be changed to meet the equipment requirements. The light tower conditions are used to construct the conditions that will be used to determine the state of each light on the light tower. See "Creating light tower " for the syntax of the conditions or rules. The configuration parameters in this branch are also used in the Operator Interface.

Name	Type	Description
xxx – Condition name	StringType	xxx – Condition description

3.2.5.5 Branch: ECS.EFEM.LightTower.Signals.Signalx

These configuration parameters are needed by each light on the light tower. The numbers must be consecutive and start with 1.

Name	Type	Description
AllowedStates	StringType	Comma separated list of signal states allowed for this signal. Valid signal states are On, Off, and Blink.
BlinkConditions	StringType	Comma separated list of condition names to be ORed together for the equation for this signal state.

Name	StringType	Name of the light. Normally corresponds with color. E.g. “Red”, “Yellow”
OffConditions	StringType	Comma separated list of condition names to be ORed together for the equation for this signal state
OnConditions	StringType	Comma separated list of condition names to be ORed together for the equation for this signal state
Present	BoolType	Component is installed

3.2.5.6 Branch: FactoryAutomation.LoadPorts

Name	Type	Description
AutoClamp	BoolType	Used to automatically clamp the FOUP at the load port and advance the carrier to the first stage of verification without user interaction.
AutoClampDelay	IntType	Time in seconds between carrier placement and automatic clamping
AutoProceedWithCarrier	BoolType	Used to advance the carrier through the verification process without host or user interaction
E84TP1Timeout	IntType	E84 TP1 Timeout in seconds
E84TP2Timeout	IntType	E84 TP2 Timeout in seconds
E84TP3Timeout	IntType	E84 TP3 Timeout in seconds
E84TP4Timeout	IntType	E84 TP4 Timeout in seconds
E84TP5Timeout	IntType	E84 TP5 Timeout in seconds
LPxIDReaderAvailable	BoolType	ID reader is enabled for the load port
PlacementTimeout	IntType	Time in seconds between present and placed signals before triggering an alarm
SingleButtonManualTransfer	BoolType	True = single button transfer, False = two button transfer

3.2.5.7 Branch: System

Name	Type	Description
EquipmentName	StringType	The name of the equipment used as the epj file base name and CIMPortal model name

3.2.6 Samples

3.2.6.1 Creating and updating variables

At CCF startup, the data models for both CIMPortal and CIMConnect will be automatically updated with the parameters defined by the software components during their creation.

To add a parameter, call the Factory Automation Client's RegisterVariables method from the constructor and supply one or more VariableDefinition instances. A VariableDefinition is used to specify the following for each new parameter:

- Name – A name for the new parameter. This name must be unique across CCF and usually includes a name, for example: "PM1.SineWave". The CxPath.Combine method, which uses the '.' character as a delimiter, is recommended for combining the elements of parameter names into a single string. The parameter name is used to determine where in the CIMPortal hierarchy model the parameter will be added. For example, a parameter named "LL1.GV.oSetOpen" would be added to the node whose CIMPortal locator would be Equipment\LL1\GV.
- VariableType – Variable type. StatusVariables have values that are valid all the time; most variables are of this type. DataVariables are variables that only have valid values when associated with an event.
- DataType – The data type of the new parameter. The following data types are supported: Short, Int, Long, Float, Double, Boolean, String, and StringArray.
- Description – A human-readable description of the parameter.
- PublishToCimConnect – Specifies whether this parameter's data values will be published through CIMConnect. All parameters are published through CIMPortal.

The following code sample shows how to create a new VariableDefinition, and how to register the new parameter using the Factory Automation Client's RegisterVariables method.

```
VariableDefinition sineWaveDefinition = new VariableDefinition(
  CxPath.Combine(this.Name, "SineWave"),
  VariableType.StatusVariable,
  DataType.Int,
  "Sine Wave",
  true);
FactoryAutomationClient.Service.RegisterVariables(
  new VariableDefinition[] { sineWaveDefinition });
```

Note that the RegisterVariables method must be called in the constructor. If the class derives from the ActiveObject class, the call must be made prior to the ActiveObject.Start() method.

The publisher of the data variables simply needs to construct an array of data variable names and an array of the data variable values. This is demonstrated in the following code:

```
string[] names = { "ToolSupervisor.Version" };
object[] values = { CxVersion.GetAllVersions() };
FactoryAutomationClient.Service.SetVariables(names, values);
```

Typically, a set of data variables is always updated at the same time. In this case, it may be worthwhile to create the name array once and then reuse it for each update. The same value array could also be reused.

3.2.6.2 Creating and using events

At CCF startup, the data models for both CIMPortal and CIMConnect will be automatically updated with the events defined by the software components during their creation.

To add an event, call the Factory Automation Client's RegisterEvents method from the constructor and supply one or more EventDefinition instances. An EventDefinition is used to specify the following for each new event:

- Name – A name for the new event. This name must be unique across CCF and usually includes the name, for example: "PM1.ProcessingStepStarted". The CxPath.Combine method, which uses the '.' character as a delimiter, is recommended for combining the elements of parameter names into a single string.
- Description – A human-readable description of the event.
- PublishToCimConnect – Specifies whether this event will be published through CIMConnect. All events are published through CIMPortal.
- DataVariableNames (optional) – Names of data variables associated with event. All data variables should be added using the method described in "Creating and updating variables". These data variables usually have the VariableType of DataVariable. If there are no data variables associated with the event, this parameter can be null.

The following code sample shows how to create a new Event and how to register the new event using the Factory Automation Client's RegisterEvents method.

```
EventDefinition processingStepStarted = new EventDefinition(
  CxPath.Combine(this.Name, "ProcessingStepStarted"),
  "Event that idicates that a processing step has just started",
  true);
FactoryAutomationClient.Service.RegisterEvents(
  new EventDefinition[] { processingStepStartedDefinition });
```

Note that the RegisterEvents method must be called in the constructor. If the class derives from the ActiveObject class, the call must be made prior to the ActiveObject.Start() method.

The publisher of the event simply needs to pass the name of the event into the TriggerEvent method. If there are associated data variables, construct an array of data variable names and an array of the data variable values, otherwise just pass in null values for the variable arrays. The TriggerEvent method demonstrated in the following code:

```
string eventName = CxPath.Combine(name, "Start");
string[] variableNames = { "CID", "Port" };
object[] variableValues = { CarrierId, location.PortId };

FactoryAutomationClient.Service.TriggerEvent(eventName, variableNames,
variableValues);
```

Typically, events do not happen on a frequently enough to warrant storage of the arrays or event name.

3.2.6.3 Adding a substrate location

A substrate location is a location in the equipment where a substrate can located. Each substrate location must have a unique name and a location type. If a module only has one substrate location, the module name should be used as substrate location. For example, if PM1 can only hold a single substrate, the substrate location name should be "PM1". For multiple substrate modules, the substrate location names should be a period ('.') delimited name with the module name first, then some descriptive term which distinguishes the different locations. Robots with multiple arms should use some type of arm descriptor, e.g. "Robot.UpperArm" or "Robot.Arm1" or "Robot.ArmA". Modules with multiple slots should use the "Slotx" or "Slotxx", for example

"LL.Slot1". For consistency, slot number should always be numbered consecutively, with Slot1 being the lowest slot.

A non-load port substrate location is added by creating a SubstrateLocationData object, then calling FactoryAutomationClient.Service.AddSubstrateLocation, as follows:

```
        SubstrateLocationData sld = new SubstrateLocationData(name,
LocationType.Aligner);
        FactoryAutomationClient.Service.AddSubstrateLocation(sld);
```

Load port substrate locations are created when load ports are added. Load ports are added directly to the FA server as follows:

```
FactoryAutomationServer.Service.AddLoadPort(i)
```

3.2.6.4 Moving a substrate

When a substrate is moved from one substrate location to another, update the FA package by calling the MoveSubstrate method as follows:

```
FactoryAutomationClient.Service.MoveSubstrate(substrate, locationId);
```

3.2.6.5 Updating substrate object model

Two methods are provided for changing the substrate object model.

One is for changing the substrate transport state. It is used as follows:

```
FactoryAutomationClient.Service.SetSubstrateTransportState(substrate, newValue);
```

The second is for changing the substrate processing state. It is used as follows:

```
FactoryAutomationClient.Service.SetSubstrateProcessingState(substrate, newValue);
```

3.2.6.6 How to use equipment performance tracking

To track equipment performance, each module to be tracked needs to be registered with the FA package. Module registration is done as follows:

```
FactoryAutomationClient.Service.AddEptModule(name, isProcessModule)
```

When a module is busy doing a task, FA should be notified as follows:

```
FactoryAutomationClient.Service.SetEptBusy(name, task, taskType);
```

When a module completes a task, FA should be notified as follows:

```
FactoryAutomationClient.Service.SetEptIdle(name);
```

If a fault condition is preventing a module from starting or completed a task, FA should be notified as follows:

```
FactoryAutomationClient.Service.SetEptBlocked(name, reason, reasonText);
```

To set an EPT state for the entire equipment, use an empty string for the name. The empty EPT name is created automatically and does not need to be added.

3.2.6.7 How to use job management

Control jobs and process jobs can be created by either the equipment operator or the host. Control jobs are managed internally by the FA package. Process job management is shared by the FA

package and the software package executing the process jobs, which is called “scheduler” in this sample. Once a process job becomes ready to execute, the FA server triggers its “ProcessJobsStarted” event. The application hosting the FA server typically registers for this event and then sends the started process jobs to the scheduler. Note that the same process job may appear in multiple “ProcessJobsStarted” events.

The FA server application host can register for the “ProcessJobsStarted” event with code similar to the following:

```
faServer.ProcessJobsStarted += new
EventHandler<ProcessJobsStartedEventArgs>(FAServer_ProcessJobsStarted);
```

A typical event handler is shown below:

```
void FAServer_ProcessJobsStarted(object sender, ProcessJobsStartedEventArgs e)
{
    if( e == null || e.ProcessJobs == null || e.ProcessJobs.Count == 0 )
        return;
    scheduler.AddJobs(e.ProcessJobs, e.Recipes);
}
```

The FA package expects to be notified when a process job should be transitioned to the “Process Completed”, “Process Job Completed”, “Paused”, “Stopped”, and “Aborted” states. The scheduler can also initiate a transition into the “Pausing”, “Stopping”, and “Aborting” states. This is done using the FA client service as follows:

```
FactoryAutomationClient.Service.SetProcessJobState(name, newState);
```

The default behavior of FA server is to send all the process jobs in a control job to the scheduler at one time. The next control job would be started when the first control job was completed. This behavior is inefficient for most equipment. It is usually desirable to start the process jobs in the next control job prior to completing the first control job. Starting the next control job’s process jobs is done using the FA server as follows:

```
    faServer.StartNextProcessJobs();
```

The equipment operator and the host can pause, stop, or abort process jobs. When a process job is paused, stopped, or aborted, the FA server triggers its “ProcessJobCommand” event. The application hosting the FA server typically registers for this event and then sends the process job command to the scheduler.

The FA server application host can register for the “ProcessJobCommand” event with code similar to the following:

```
faServer.ProcessJobCommand += new
EventHandler<ProcessJobCommandEventArgs>(FAServer_ProcessJobCommand);
```

A typical event handler is shown below:

```
void FAServer_ProcessJobCommand(object sender, ProcessJobCommandEventArgs e)
{
    if( e == null || e.ProcessJob == null )
        return;

    switch( e.Command )
    {
        case E40PJMLib.CommandCallbackEnum.pjcbPAUSE:
```

```
            if( scheduler.HasJob(e.ProcessJob.Name) )
                scheduler.PauseJob(e.ProcessJob.Name);
            else
                FactoryAutomationClient.Service.SetProcessJobState(
e.ProcessJob.Name, E40PJMLib.ProcessJobStateEnum.pjPAUSED);
            break;
        case E40PJMLib.CommandCallbackEnum.pjcbSTOP:
            if( scheduler.HasJob(e.ProcessJob.Name) )
                scheduler.StopJob(e.ProcessJob.Name);
            else
                FactoryAutomationClient.Service.SetProcessJobState(
e.ProcessJob.Name, E40PJMLib.ProcessJobStateEnum.pjSTOPPED);
            break;

        case E40PJMLib.CommandCallbackEnum.pjcbABORT:
            if( scheduler.HasJob(e.ProcessJob.Name) )
                scheduler.AbortJob(e.ProcessJob.Name);
            else
                FactoryAutomationClient.Service.SetProcessJobState(
e.ProcessJob.Name, E40PJMLib.ProcessJobStateEnum.pjABORTED);
            break;
    }
}
```

3.2.6.8 How to create a custom FactoryAutomationServer

In the project specific code, create a new factory automation server class (called MyFactoryAutomationServer in this example). This project class must be derived from FactoryAutomationServer. Implement the desired functionality in the new class. The application hosting the factory automation server should be modified to create the new project factory automation server.

Do this by changing:

```
faServer = new FactoryAutomationServer();
```

To:

```
faServer = new MyFactoryAutomationServer();
```

3.2.6.9 How to create a custom FactoryAutomationClientService

In the project specific code, create a new factory automation client service class (called MyFactoryAutomationClientService in this example). This project class may be derived from FactoryAutomationClientService, or may implement the IFactoryAutomationClientService interface. Implement the desired functionality in the new class.

An example deriving from FactoryAutomationClientService would be:

```
class MyFactoryAutomationClientService : FactoryAutomationClientService
{
    public override IExecutionAlarm CreateExecutionAlarm(string name)
    {
        return new MyExecutionAlarm(name);
```

```
    }
}
```

Each project application that uses the factory automation client (including the application hosting the factory automation server) should set the Service property on the FactoryAutomationClient class before starting to use the factory automation client service.

```
FactoryAutomationClient.Service = new MyFactoryAutomationClientService();
```

3.2.6.10 How to use WCF to connect FactoryAutomationServer and FactoryAutomationClientService

The CCF FactoryAutomationClientService does not need to be in the same application as the CCF FactoryAutomationServer. They can communication with each other using WCF (Windows Communication Foundation).

To configure the CCF FactoryAutomationClientService to use WCF to communicate with the FactoryAutomationServer, information must be added to the configuration file of the application with the FactoryAutomationClientService. In a Visual Studio project, this file is usually named "App.config". After building the project, the "App.config" file is copied to the output folder and renamed "xxxx.exe.config", where xxxx is the application name. To use the TCP/IP transport layer, the client application should have an entry similar to the following, where localhost is the name of the computer hosting the FactoryAutomationServer and 6408 is the port number on which the server is listening:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <bindings>
      <netTcpBinding>
        <binding name="CcfDefaultBinding" transferMode="Buffered"
maxConnections="30"
          maxReceivedMessageSize="2147483647">
          <readerQuotas maxArrayLength="2147483647" />
          <security mode="None">
            <transport clientCredentialType="None" protectionLevel="None" />
            <message clientCredentialType="None" />
          </security>
        </binding>
      </netTcpBinding>
    </bindings>
    <client>
      <endpoint address="net.tcp://localhost:6408" binding="netTcpBinding"
          bindingConfiguration="CcfDefaultBinding"
contract="Cimetrix.CimControlFramework.FactoryAutomation.IFactoryAutomationServer"
          name="FactoryAutomationEndpoint" />
    </client>
  </system.serviceModel>
</configuration>
```

To configure the CCF FactoryAutomationServer to use WCF to communicate with the alarm client services, information must be added to the configuration file of the application host the FactoryAutomationServer. To use the TCP/IP transport layer, the server application should have an

entry similar to the following, where localhost is the name of the computer hosting the FactoryAutomationServer and 6408 is the port number on which the server should listen:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <bindings>
      <netTcpBinding>
        <binding name="CcfDefaultBinding" transferMode="Buffered"
maxConnections="30"
          maxReceivedMessageSize="2147483647">
          <readerQuotas maxArrayLength="2147483647" />
          <security mode="None">
            <transport clientCredentialType="None" protectionLevel="None" />
            <message clientCredentialType="None" />
          </security>
        </binding>
      </netTcpBinding>
    </bindings>
    <services>
      <service name="
Cimetrix.CimControlFramework.FactoryAutomation.FactoryAutomationServer">
        <endpoint address="net.tcp://localhost:6408" binding="netTcpBinding"
            bindingConfiguration="CcfDefaultBinding"
contract="Cimetrix.CimControlFramework.FactoryAutomation.IFactoryAutomationServer "
/>
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

Note that the bindings on the client and server sides must match. If a custom factory automation server is being used, the service name attribute "Cimetrix.CimControlFramework.FactoryAutomation. FactoryAutomation Server" must be changed to match the namespace and class name of custom factory automation server.

In addition, the server application must call the StartServiceHost method on FactoryAutomationServer.