Study Guide-Modules and Components

Title	Study Guide-Modules and Components
URL Name	Study-Guide-Modules-and-Components
Description	

Abstract

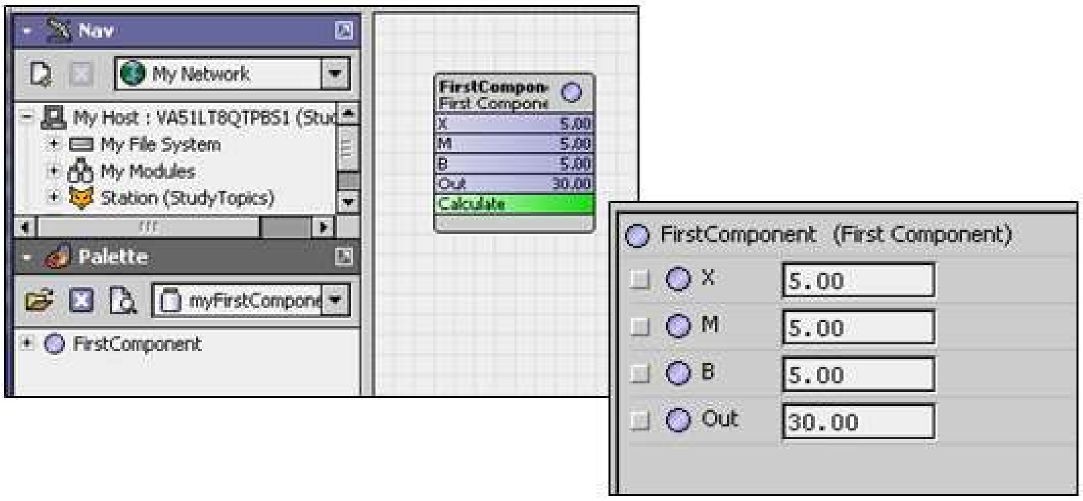
The Niagara Framework is made up of a collection of distributable JAR files known as Modules. Modules allow third party developers to extend the Niagara Framework with domain specific functionality to create new product features that customers may add to their Niagara installations.

A Module is a collection of compiled Java code, packages, and Module descriptor files, along with any other resources required for the compiled code to run correctly.

The Niagara framework makes use of the BObject model to normalize data into immutable data values referred to as Simples and complex data structures which may be recursively broken down into these simple data value types. These complex data structures come in the form of BComponents and BStructs. Components and Structs are a collection of Slots which allow these objects to seamlessly integrate with components and structs developed by other engineers and be "wired" together through the workbench wiresheet view. Slots provide the basis for "Component Oriented Programming."

This Study Guide covers in depth the creation and components of a Module. It also covers the BObject model and the creation of a BComponent with Property and Action slots. The completion of the exercises of this study guide should result with a module and BComponent accessible in the Niagara Workbench and added to a running station.

Contents



Prerequisites

This guide assumes familiarity with the Java programming language, Object Oriented concepts, and with using the Niagara workbench.

Compatibility
Niagara AX 3.4, 3.5, 3.6, 3.7, 3.8

Previous Study Guides
None

Module Dependencies
baja

Baja Docs
BComponent, Property, Action, Topic, Type

Developer Docs
Object Model, Component Model, Modules, Build, Building Complexes

The Niagara Plugin

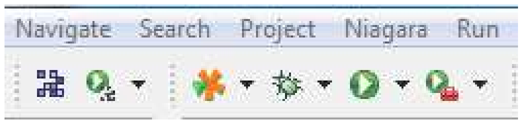
The Eclipse IDE uses Plugin jar files in a manner similar to Niagara modules to extend the Eclipse platform. The Niagara Plugin jar file provides Niagara specific application development functionality within the Eclipse framework. Tools such as slot-o-matic and the Niagara build tool may be accessed from the Eclipse environment once the Plugin is installed.

The Niagara Plugin requires that the Niagara Framework be installed on the same platform that is used to run the Eclipse IDE. Multiple Niagara installations may be installed on the same developer platform. The Niagara Plugin requires that a Niagara installation is selected as the current Niagara AX Home target to compile against.

Exercise: Installing the Niagara Plugin

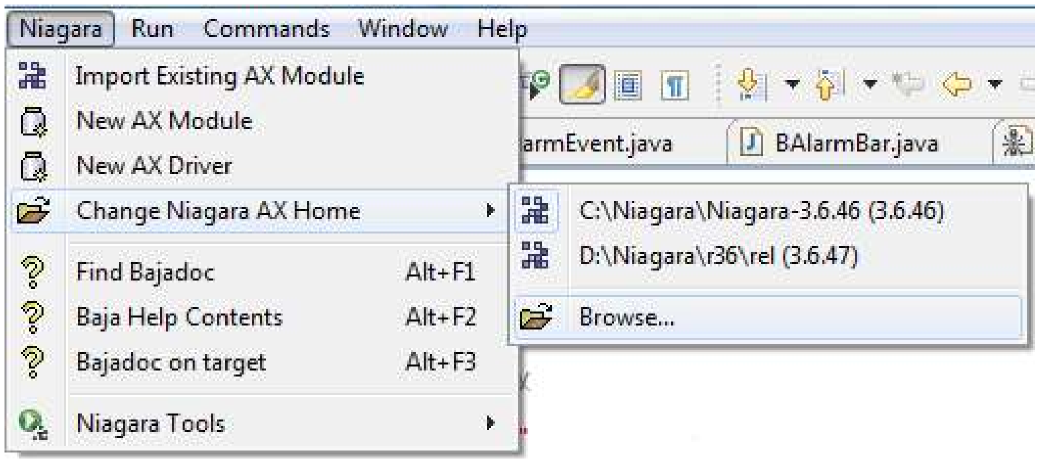
1. Ensure that the Eclipse IDE is installed on your computer. The Eclipse Niagara Plugin is supported in version 3.3, 3.4, 3.5, and 3.6 of Eclipse. To install Eclipse, go to the Eclipse download website and download and install the Eclipse Classic download.
2. If you have not already done so, download the latest Niagara Eclipse Plugin jar file from the Niagara Central portal.
3. In the Eclipse installation directory, drop the Niagara Plugin into the Plugins directory.
4. Open the Eclipse IDE. Search the Menu bar and Tool bar at the top of the IDE. You should see the Niagara menu and Niagara tools present.

The Niagara Eclipse Plugin Toolbar



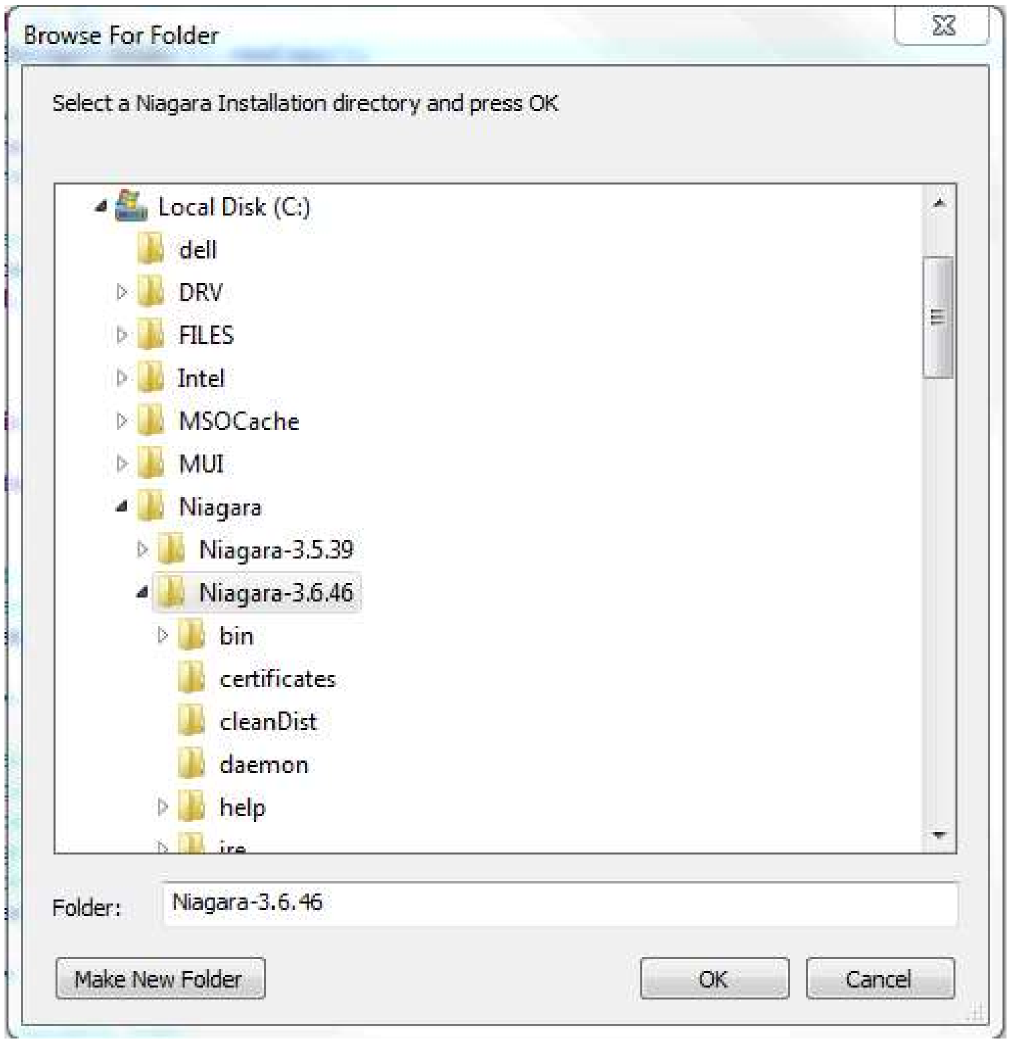
5. Click on the Niagara menu bar and select Change Niagara AX Home. If other installations have been selected previously, they will display in the menu and may be selected as the current Niagara AX target. To select a new Niagara installation, click the Browse button.

Niagara AX Home Path Menu Option



6. Using the Browse For Folder dialog window, browse to a Niagara installation. The directory should be a top level directory with a naming format of Niagara-3.X.xx.

Niagara Home Directory Browser



Once the Niagara directory is selected, a dialog will display indicating that the Niagara Plugin is booting in the Eclipse environment.

On completion of installing the Niagara Plugin, tools such as the Build tool and Slot-o-matic are available for use through the Eclipse environment.

Modules and the New Module Wizard

The Niagara Framework is a collection of distributable modules. A module is a deployable versioned JAR file that contains compiled Java class files along with any other resources required for the code to properly execute. Modules plug into the Niagara Framework and provide compartmentalization of domain specific features.

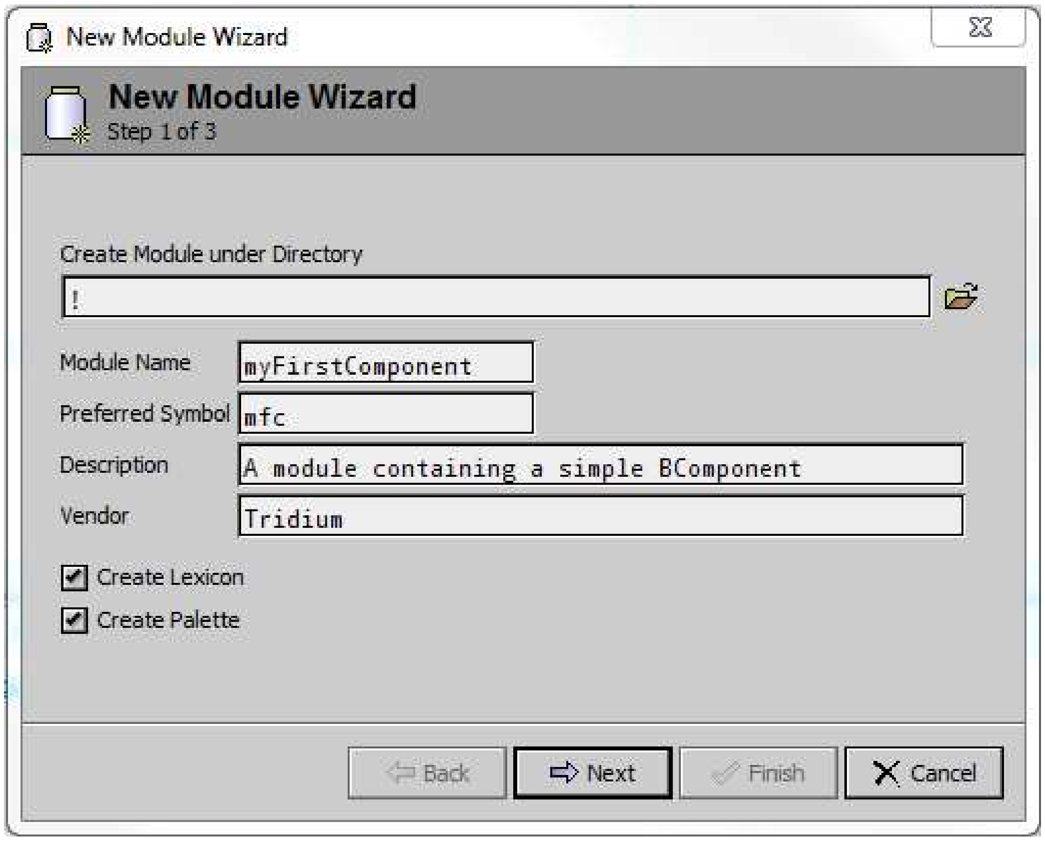
From a practical stand point, modules provide third party developers a means to extend and build on top of the Niagara Framework. Applications intended to run on the Niagara Framework are distributed through modules.

Modules can be created simply through the Module Wizard. The Module Wizard is accessible through either the Niagara Workbench or through the Niagara Eclipse Plugin.

When the module wizard is executed, a module directory will be created at the specified file path with a root src directory and two or more module descriptor files.

The first page of the New Module wizard includes several fields used to specify how the module should be created.

The New Module Wizard



Create Module under Directory: !

Module Name: myFirstComponent

Preferred Symbol: mfc

Description: A module containing a simple BComponent

Vendor: Tridium

[✓] Create Lexicon

[✓] Create Palette

The **Directory** field includes a default path using the bang (!) character. This character indicates the install directory of the Niagara installation currently selected as the **Niagara AX Home** installation for the Niagara Plugin. This path can be set to any valid directory path desired. Clicking the folder icon next to directory field displays a directory browser dialog that can also be used to select the path of the module directory.

The **Module Name** field provides the name to use for the directory that the module files are created in as well as the name of the module as defined in the build.xml file (discussed later). When the module is built into a JAR file, the JAR file will use the module name as the JAR file name.

The **Preferred Symbol** field is used to provide a shorthand symbol of the module name to uniquely identify the module in the Niagara Framework's Type system. The Type system is not covered in this Study Topic, but is discussed in other Study Topics.

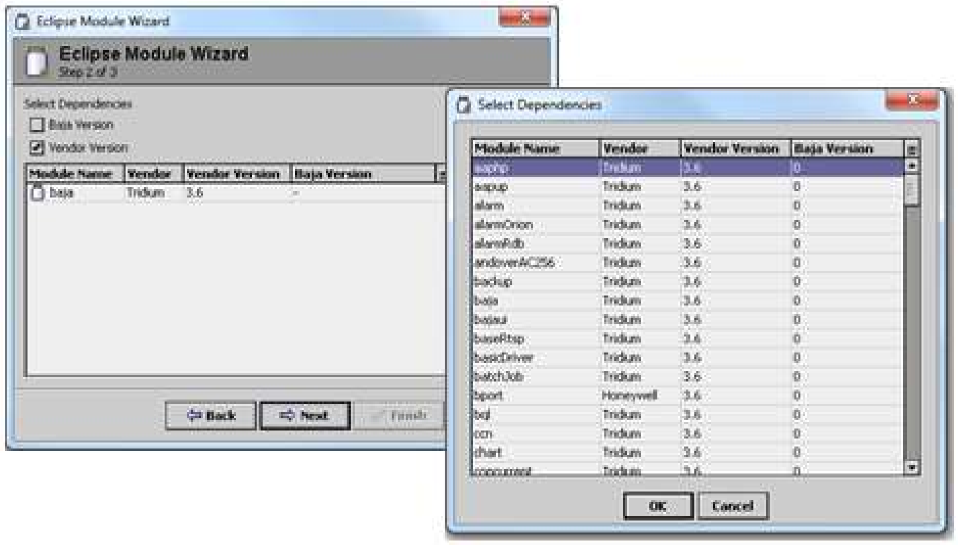
The **Description** field and **Vendor** field are provided to assist System Integrators in selecting which modules to install onto a system. The description field provides a short description of the module. The vendor field provides the name of the vendor through which the module is distributed.

The **Create Lexicon** and **Create Palette** checkboxes determine whether the optional **module.lexicon** and **module.palette** files are created for the module. Checking these fields will create these optional files in the final generated module directory.

One field which has not been discussed is the **Baja Version** field. This field should be left at the default value 0. This field is reserved for incrementing the Baja specification version. The Baja specification has yet to be revised, and as such this field should not be modified.

The next page of the New Module wizard is used to configure the dependencies of the module.

Module Dependency Setup



Eclipse Module Wizard

Eclipse Module Wizard
Step 2 of 3

Select Dependencies

- [ ] Baja Version
- [✓] Vendor Version

Module Name	Vendor	Vendor Version	Baja Version
baja	Tridium	3.6	-

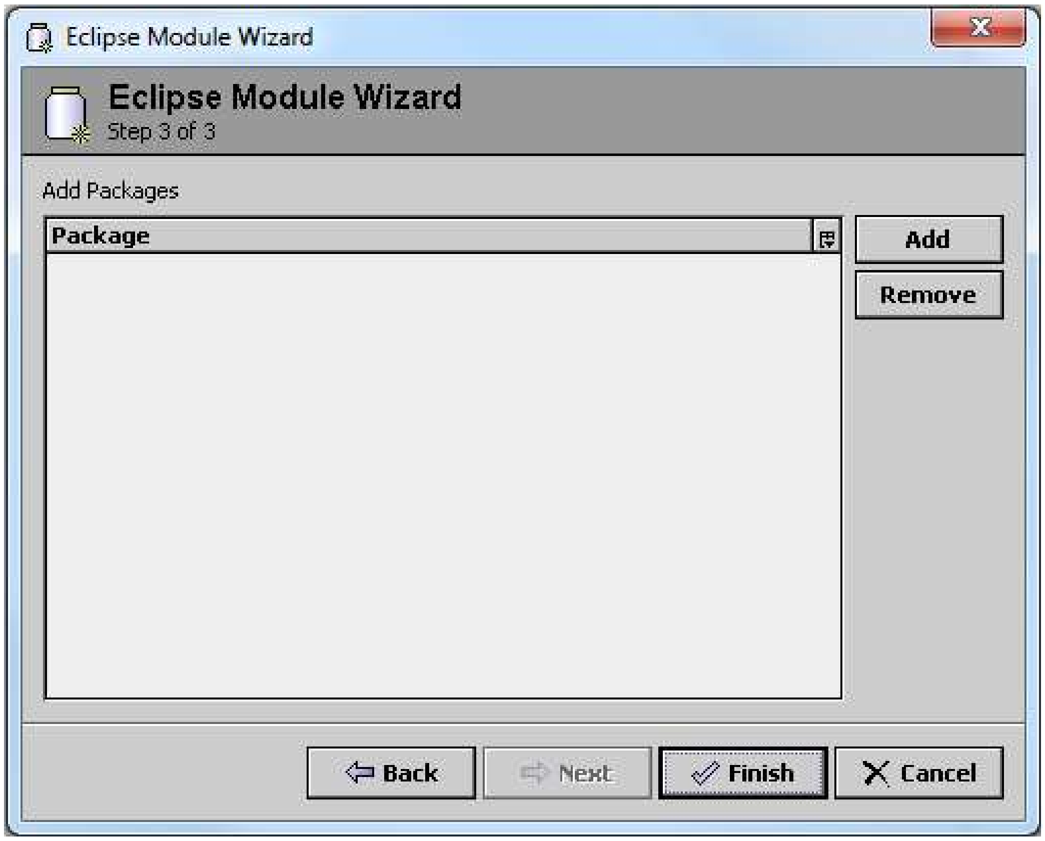
Select Dependencies

Module Name	Vendor	Vendor Version	Baja Version
aaphp	Tridium	3.6	0
aapup	Tridium	3.6	0
alarm	Tridium	3.6	0
alarmOrion	Tridium	3.6	0
alarmRdb	Tridium	3.6	0
andoverAC256	Tridium	3.6	0
backup	Tridium	3.6	0
baja	Tridium	3.6	0
bajaui	Tridium	3.6	0
baseRtsp	Tridium	3.6	0
basicDriver	Tridium	3.6	0
batchJob	Tridium	3.6	0
bport	Honeywell	3.6	0
bql	Tridium	3.6	0
ccn	Tridium	3.6	0
chart	Tridium	3.6	0
concurrent	Tridium	3.6	0

Just like Java classes use the **import** statement to create dependencies on other compiled class code files, modules may use objects defined in other modules. Dependencies can be added through the **Select Dependencies** dialog. When the module is generated, the module dependencies will be included in the module�s build.xml file.

The final page of the wizard is used to declare packages within the module.

Module Wizard Package Declaration



Packages are directories within the module that are used to namespace Java source files and resources. When a Java package is created, it must also be included in the build.xml file to be compiled into the final module jar file. This is discussed later in the build.xml study section.

Exercise: Creating a Module

1. In the Eclipse environment, select the Niagara menu. From the Niagara menu, several sub-menu options are available and will be discussed in more detail later on. Select the **New AX Module** menu option to begin creating a new Module.
2. Set the path directory to the bang (!) character. This is the install directory currently selected Niagara AX Home installation for the Niagara Plugin. You may alternately set the path to a directory of your choosing. This is the directory which will store all module files and which will be used by the Eclipse IDE to create a Java project.
3. Set the Module Name to **myFirstComponent**. The preferred symbol field should be set to mfc. Check the Create Lexicon and the Create Palette checkboxes to ensure these optional files are generated with the module.

Field	Value
Module Name	myFirstComponent
Preferred Symbol	mfc
Vendor	{ Your Company Name }
Create Palette	Checked
Create Lexicon	Checked

4. Once the module fields are completed click the **Next** button. The Dependencies setup page will display. For this exercise, we will not be using additional dependencies other than the baja module.
5. Click the **Next** button and then click **Finish**. In the Package Explorer view of the Eclipse IDE, verify that a new Java Project named MyFirstComponent is created.
6. Open the project directory from the Package Explorer and verify that the following files and directories exist:

- build.xml
- module-include.xml
- module.lexicon
- module.palette
- src folder directory

Types and the BObject Model

At the heart of the Niagara Object Model is the Type System layered above the Java Type system. The Type system allows the Niagara Framework to uniquely identify a Niagara Object across the framework. All Niagara objects declare their type as a public static final TYPE field in the body of the class file.

Every type is globally defined by its TypeSpec, which uses the following format:

```
{module name}:{type name}
```

The type name is the name of the class without the requisite B letter prefixing the class name. An example for a Type Spec of a BObject is seen in the BAbsTime Type Spec:

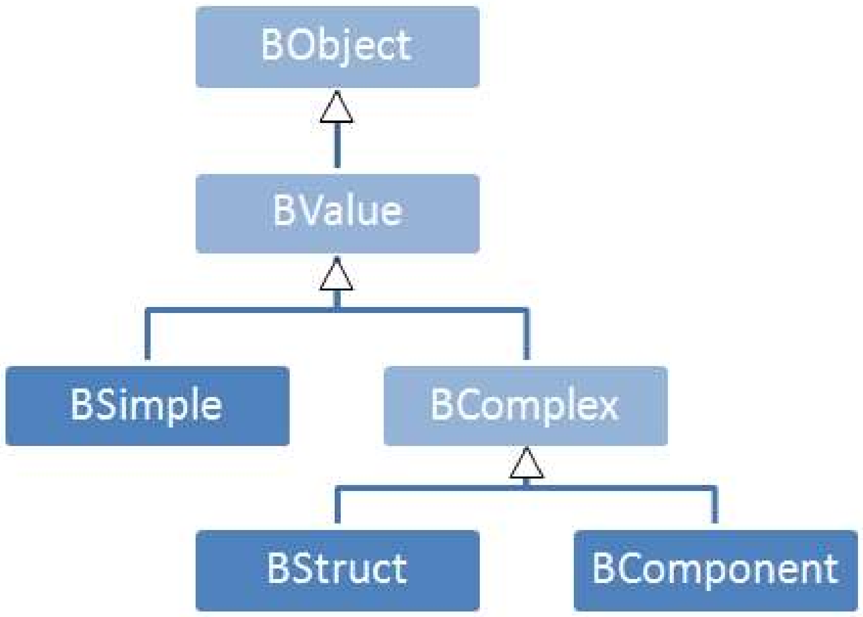
```
BAbsTime <-----> baja:AbsTime
```

Types are mapped to a qualified Java class name in the module-include.xml file. Each BObject has a Type entry in the module include as follows:

```
< type name="FooBar" class="javax.baja.control.BFooBar" />
```

All Java classes which implement a Niagara Type are subclassed from BObject. The BObject Model is the hierarchy of subclasses on which the Niagara Framework is built.

The BObject Model



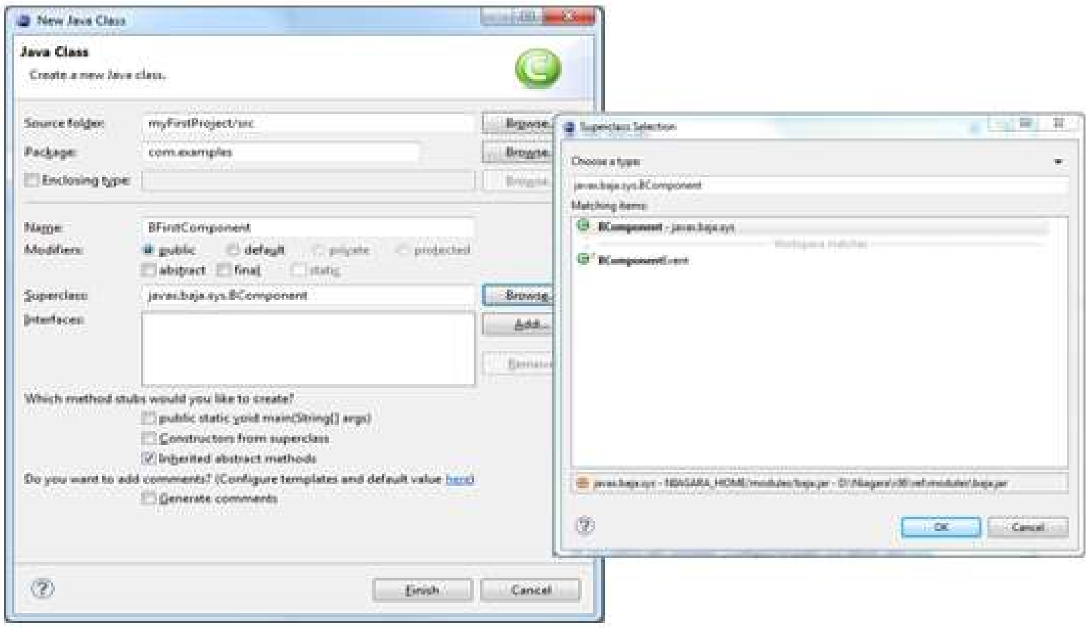
The base classes of the BObject model are contained in the **javax.baja.sys** package as part of the Niagara Framework public API. Classes which extend these base classes may exist in other packages. While BSimple objects represent primitive data pieces that cannot be decomposed, objects extending from BComplex are a collection of Slots which can be recursively broken down into primitive BSimple values.

Exercise: Creating a BComponent

1. In the Eclipse environment, select the Project folder created by the New Module Wizard in the Package Explorer tree view.
2. Right click on the src directory folder of the Java Project. From the pop-up menu, select **New -> Class**. The New Java Class Wizard will display.

   The Eclipse New Java Class Wizard allows the developer to specify the package, name, super class, and interfaces of a new class. When declaring the package name, if the package does not exist in the project then Eclipse will create the package structure in the Java project.

   Eclipse New Java Class Wizard


New Java Class

Java Class

Create a new Java class.

Source folder: myFirstProject/src

Package: com.examples

Enclosing type: [ ]

Name: BFirstComponent

Modifiers: [✓ public] default private protected; abstract [ ] final [ ] static [ ]

Superclass: javax.baja.sys.BComponent

Interfaces:

Which method stubs would you like to create?

[ ] public static void main(String[] args)

[ ] Constructors from superclass

[✓] Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

[ ] Generate comments

Superclass Selection

Choose a type: javax.baja.sys.BComponent

Matching items:

BComponent - javax.baja.sys

Workspace matches

BComponentEvent

javax.baja.sys - NIAGARA_HOME/modules/baja.jar - D:\Niagara\r38\ref\modules\baja.jar

Caution

When defining the Superclass of a Java Class, the New Java Class Wizard provides a Superclass Selection dialog to quickly select a super class from the project's referenced libraries. Click the Browse button next to the Superclass field and type in the name of the class to extend in the **Choose a Type** field. Eclipse will provide the closest matching options of available classes.

3. In the new Java Class Wizard dialog, set the dialog fields as follows:

Field	Value
package	com.examples
name	BFirstComponent
Superclass	javax.baja.BComponent

4. Once you have completed filling out the class information in the class wizard dialog, click the Finish button. The dialog will close and the Eclipse editor will display the new java file. Note that Eclipse will auto generate any required methods inherited from the super class or implemented interfaces with NULL or default values.

Note

The Niagara Plugin automatically includes boiler plate comment header code for use with Slot-o-matic. This comment code can be ignored for this study guide.

5. In the class body, create a code comment block. The comment should simply read *Type*. Beneath the comment block, include a getType() function and a static final Type variable named TYPE. The constant will use the Sys.loadType method to statically load the Type for your class.

```
////////////////////////////////////////////////////////////
// Type
////////////////////////////////////////////////////////////

  public Type getType() { return TYPE; }
  public static final Type TYPE =
                        Sys.loadType(BFirstComponent.class);
```

6. Import the javax.baja.sys.Sys and javax.baja.sys.Type classes. Alternatively, you can use the Eclipse shortcut Ctrl + Shift + o to automatically include the appropriate import statements.

7. Open the module-include.xml file in the Eclipse editor. Add the Type information for your BComponent within the <types> root element as shown.

```
<types>
  <type name="FirstComponent" class="com.examples.BFirstComponent"/>
</types>
```

8. Save the file changes and close the module-include file.

Slots

In the Niagara framework, BComponents are a collection of Slots which define the Properties, Actions, and Topics available for the component. Using Slots to define the information, behavior, and events of a BComponent creates an environment for Component Oriented Programming. This in turn allows components to seamlessly integrate into the framework, making them available for use by other components. Component Oriented Programming allows components to be "wired" together by non-programmers in the wiresheet view of the workbench to create station logic.

Slots are objects stored on a BComponent or BStruct object. Every child slot of a BComponent or BStruct is identified by a String name unique within the parent component. Slots include a 32 bit-mask flag value to identify special purposes of the slot and a map of name-value pairings (facets) that include metadata about the slot or the slot's current value.

There are three types of slots: Property slots, Action slots, and Topic slots.

A Property slot stores a BValue object that is accessible through getter and setter methods on the component. The Property slot defines the State information of a component. Property slots require a default value to use when the component is first instantiated. This ensures that the slot always contains a value other than NULL. By default, property slot values are serialized and persisted to the station BOG file at time of station shutdown.

An Action slot is a behavior that may be invoked on the component. This behavior can be triggered by a topic event, a programmatic call, or directly by a user through the workbench. An action may return a BValue and may define a single parameter argument or no argument at all. Actions also require a special callback implementation in the code which will be described in the next exercise. Actions invoked in the proxy space (client side workbench) execute in the Master space (station).

Topics broadcast event information to subscribing objects. Topics are fired using a fire[TopicName] invocation pattern. Firing a topic on a component in the proxy space will fire the same topic on the corresponding component in the master space.

Slot Type	**Description**
Property	The State information of the component or struct.
Action	A behavior which may be invoked on the component.
Topic	An event that may be broadcast by the component.

One last note about slots: slots defined in code (referred to as Frozen Slots which are discussed in greater detail in later guides) are defined as static final objects on the parent object. Slots must be defined before the Type declaration of the

BComplex object.

Exercise: Defining Slots on a BComponent

For this exercise, our component will use the values of three different properties to perform a calculation. The calculation will be executed as an action manually invoked by the end user. Both the input properties and the output property will be of type **double**.

1. Reopen the ***BFirstComponent*** java source file if it is not already displayed in the Eclipse editor. Above the Type declaration declare a ***public static final Property*** class variable named **x**. Set the value as the return value of the inherited static method newProperty:

```
////////////////////////////////////////////////////////////////
// Properties
////////////////////////////////////////////////////////////////

public static final Property x = newProperty(Flags.SUMMARY,5);
```

Warning

Be sure to declare all Property, Action, and Topic Slots **BEFORE** the public static final Type TYPE declaration. If you do not do this, the component slots will not be correctly introspected at time of startup and the component will fail to load in the Workbench and station!

Note

The static declaration creates the property for all instances of our new component. The value itself is stored separately, allowing for significant memory optimization by the framework. Think of this property declaration as acting like a hashmap key, which we'll use to store and retrieve the actual values for this property on each component.

Note

The Flags.SUMMARY value is a Flag assigned to the slot to make the slot visible in the Wiresheet view when we create an instance of our BComponent in our running station.

2. Create a get and set method to access the value of the property beneath the property declaration. The value stored in this property is of type double, so ensure that the get method returns a value of type double and the set method takes an argument of type double.

   Because we are using a primitive type (double), we will make use of the framework's primitive to BObject conversion method to store and retrieve the value as a double.

```
public double getX() { return getDouble(x); }

public voud setX( double v) { setDouble(x, v, null); }
```

3. Repeat step one and two to create the three following properties: m, b, and out.

   At this point, there should be four (4) property declarations in the BFirstComponent class file.

```
/**
 * Input Property X representing a value of type double. Each
 * property field is a static final declaration.
 */
 public static final Property x = newProperty(Flags.SUMMARY, 5);
 public double getX() { return getDouble(x); }
 public void setX(double v) { setDouble(x,v,null); }

 /**
  * Input Property M representing a value of type double
  */
  public static final Property m = newProperty(Flags.SUMMARY, 5);
  public double getM() { return getDouble(m); }
  public void setM(double v) { setDouble(m,v,null); }

  /**
   * Input Property B representing a value of type double
   */
  public static final Property b = newProperty(Flags.SUMMARY, 5);
  public double getB() { return getDouble(b); }
  public void setB(double v) { setDouble(b,v,null); }

  /**
   * Output Property Out representing a value of type double
   */
   public static final Property out = newProperty(Flags.SUMMARY, 5);
   public double getOut() { return getDouble(out); }
   public void setOut(double v) { setDouble(out,v,null); }
```

4. Below the final property slot declaration, ***but still above the Type declaration***, create a new public static final Action class variable named calculate. The execute action takes no argument and returns no value.

```
////////////////////////////////////////////////////////////////
// Actions
////////////////////////////////////////////////////////////////

  public static final Action calculate = newAction(Flags.SUMMARY);
```

5. Create a public calculate method that uses the inherited invoke method to invoke the action slot:

```
public void calculate() { invoke( calculate, null, null ); }
```

6. Create a callback method for our execute Action named doCalculate. This method is of type void and will set the component's out property value with the result of the slope intercept calculation mx + b.

```
/**
 * The callback made by the framework when our calculate Action
 * is invoked. By using a callback pattern, the action can be
 * invoked on the client side (Workbench) and executed on
 * the station side.
 */
 public void doCalculate()
 {
   double m = getM();
   double x = getX();
   double b = getB();

   double slopeIntercept = m*x + b;

   setOut(slopeIntercept);
 }
```

Note

This method is required and is called by the framework whenever the action is invoked. The callback gives us the opportunity to define how the component will react to the invocation. The naming of this method is important; the action invocation callback will always begin with **do** prefixed to the name of the action. Some actions may take a single parameter and return a value, although our example does not.

7. Check for syntax errors and class dependency errors. You may import the required packages manually by adding import statements to the top of the java file or simpley press Ctrl + Shift + O to organize the source

class imports automatically. Once all syntax errors and dependencies are resolved, save the file.

Warning

Make sure to use the **javax.baja.sys.Type** and **javax.baja.sys.Property** imports. Otherwise your module will fail to compile properly.

With the property slots and action slot defined and our action callback handled, the component development process is complete. Before the component can be used, we must build our module. To make our component easily accessible to an end user, we will create a palette item to allow the user to drag and drop an instance of our new component into a running station.

Palette Items

Palette items allow end users to drag and drop BObject instances from the Workbench into a running station. Palette items are provided by Modules through the module.palette file.

The Palette file is a Baja Object Graph file, or BOG file, with the same syntax as a station BOG file. The palette file is a collection of property slot elements which at the top level represent BComponent and BStruct objects.

If the value of a property slot also has property slots, those subsequent slots may also be configured with default values. When the palette item is added to a station, the component instance that the palette item represents will be instantiated with the values defined in the palette file for that item.

```
<?xml version="1.0" encoding="UTF-8"?>
<bajaObjectGraph version="1.0">
<p m="b=baja" t="b:UnrestrictedFolder">
  <p n="SineWave" m="kc:kitControl" t="kc:SineWave">
    <p n="amplitude" v="35"/>
  </p>
  <p n="Add" t="kc:Add">
    <p n="Link" t="b:Link">
      <p n="sourceOrd" v="h:1"/>
      <p n="sourceSlotName" v="out"/>
      <p n="targetSlot" v="inA"/>
    </p>
  </p>
</p>
```

Each property slot element has several attributes which may be defined.

Attribute	Description
n	**-Required-** Name of the Slot
m	Defines a module symbol using the format "symbol=name". Once defined a symbol can be used in subsequent type attributes.
t	Specifies type of property using format {symbol}:{typename}. Symbol must map to module declaration.
f	Specifies slot flags using format defined by Flags.encodeToString()
v	String encoding value of a BSimple.
x	Specifies slot facets using format defined by BFacets.encodeToString()

If the property slot element has no parent, or the property slot does not correspond with a slot defined explicitly at compile time (referred to as 'frozen') on a parent property, the property element must include a type attribute. If the property slot element name corresponds to a frozen property of the parent element, the type should not be defined.

The module symbol must always be defined before a type attribute value is set for that module.

Exercise: Adding a Palette Item

1. Double click on the module.palette file under the myFirstComponent Java Project in the Package Explorer. The palette XML file will display in the Eclipse editor.
2. Modify the palette XML code to include the following XML snippet and save the palette file:

```
<?xml version="1.0" encoding="UTF-8"?>
<bajaObjectGraph version="1.0">
<p m="b=baja" t="b:UnrestrictedFolder">

  <p n="FirstComponent" m="mfc=myFirstComponent"
                        t="mfc:FirstComponent" />

</p>
</bajaObjectGraph>
```

Warning

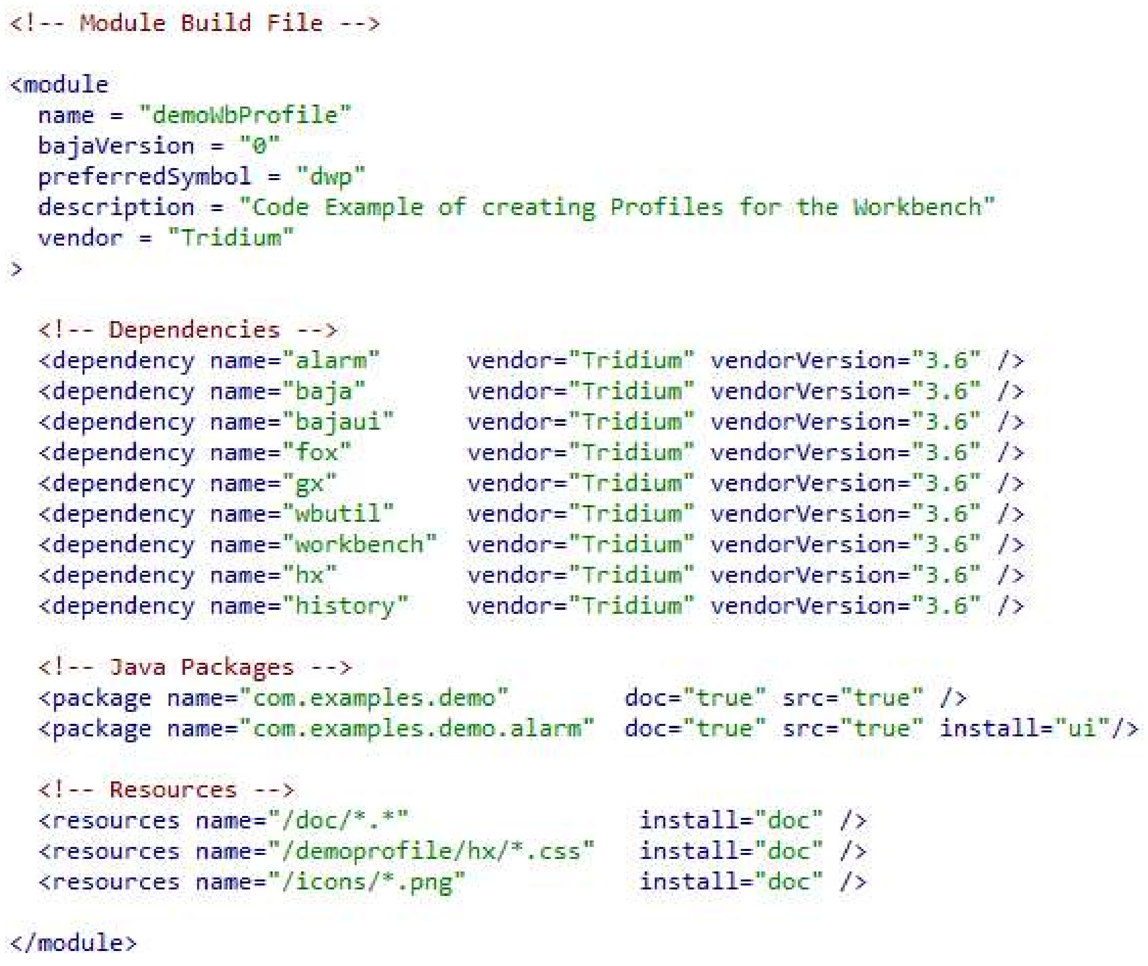
Mistakes are easy to make when manually editing a palette item! You can always edit the module.palette file through the workbench and add BObjects to the palette file through the Slot view as new property slots.

Once the palette item is created, we are ready to build our module.

The build.xml File

The build.xml file is used by the Niagara build tool to resolve how to build the module. The XML structure includes a root module element followed by zero or more dependency, package, and resource elements.

Sample build.xml File



```
<!-- Module Build File -->

<module
  name = "demoWbProfile"
  bajaVersion = "0"
  preferredSymbol = "dwp"
  description = "Code Example of creating Profiles for the Workbench"
  vendor = "Tridium"
>

  <!-- Dependencies -->
  <dependency name="alarm"      vendor="Tridium" vendorVersion="3.6" />
  <dependency name="baja"       vendor="Tridium" vendorVersion="3.6" />
  <dependency name="bajaui"     vendor="Tridium" vendorVersion="3.6" />
  <dependency name="fox"        vendor="Tridium" vendorVersion="3.6" />
  <dependency name="gx"         vendor="Tridium" vendorVersion="3.6" />
  <dependency name="wbutil"     vendor="Tridium" vendorVersion="3.6" />
  <dependency name="workbench"  vendor="Tridium" vendorVersion="3.6" />
  <dependency name="hx"         vendor="Tridium" vendorVersion="3.6" />
  <dependency name="history"    vendor="Tridium" vendorVersion="3.6" />

  <!-- Java Packages -->
  <package name="com.examples.demo"        doc="true" src="true" />
  <package name="com.examples.demo.alarm"  doc="true" src="true" install="ui"/>

  <!-- Resources -->
  <resources name="/doc/*.*"                install="doc" />
  <resources name="/demoprofile/hx/*.css"   install="doc" />
  <resources name="/icons/*.png"            install="doc" />

</module>
```

Note

The above syntax is highlighted for illustration purposes only. Other XML editors may use a different syntax coloring scheme. For a free XML editor, check the Eclipse website for an appropriate Plugin.

The <module> Root Element

The root module element describes the module contents and includes five required attributes:

Attribute	Description
name	-Required- Name of the module. This should match the module project directory name.
bajaVersion	-Required- Baja specification version number, or "0" if not a public specification.
preferredSymbol	-Required- A short symbol to use as an abbreviation for the module name. This symbol is used during XML serialization.
description	-Required- A short description of the module.
vendor	-Required- The name of the vendor for the module.

The <dependency> Element

By convention, the first child element type of the module element is the dependency element. Each dependency element declares an external module that the current module being built is dependent upon.

Modules declared as dependencies must be built before this module can be built and must match the name, vendor, and version information specified in the dependency attributes:

Attribute	Description
name	-Required- The name of the module being depended upon.
vendor	-Required- Specifies a required vendor name for the module.
vendorVersion	-Optional- Specifies the minimum required vendor version of the module on the target Niagara platform. The identified module installed on the target Niagara platform must either match this version or be a more recent release of the module.

The name of the module simply identifies the name of the module to use as a dependency. The vendor and vendorVersion are used to further uniquely identify the module to use.

The vendor version specifies which version of the module to use in the case that the vendor has released multiple versions of the same module. If a module of the specified name and vendor cannot be found with the given vendor version or a more recent version, the module will fail to load into the Niagara runtime environment of the target platform. Specifying the vendor version allows the developer to take advantage of feature enhancements or bug fixes for the specified module and ensure that these same features and fixes are available on the target platform.

The <package> Element

Modules are composed of packages containing Java source code files. The package element identifies which source code files to compile as well as the files to use for generating reference documentation in the form of Baja Docs.

The package attributes are as follows:

Attribute	Description
name	-Required- The name of the package as it would be declared in a package or import statement. This name must correspond to a directory declared within the src directory in the module's directory structure
doc	-Optional- This attribute is used to determine how documentation should be generated for the package. If set to true, both bajadocs and javadocs are generated for the source files in the package. If set to false, no documentation is generated. If set to bajaonly, then only BObject Types and slot documentation is generated.
compile	-Optional- If declared, this attribute must be either true or false. The default value is true. If false, the source code files in the package will not be compiled. Documentation may still be created from the package source code files.
install	-Optional- This attribute is used during provisioning to strip optional directories for headless devices. Valid values are "runtime", "ui", and "doc". The default is defined by the module root element.
edition	-Optional- This attribute determines which Java library edition to compile against.Valid values are "j2me", "j2se", and "j2se-5.0". The default is "j2me".

The <resource> Element

Source code within a module may reference resources such as image and icons. These resources must be packaged within the module JAR file and declared as resource elements in the build.xml file. The resource element includes a name attribute which specifies the path to the resource file or files in relation to the src directory of the module and may make use of wildcard characters such as '*.*' to specify multiple files.

Attribute	Description
name	-Required- Name of the file or files to copy from the "src" to "libJar" directory. The path is relative to the "src" directory and is copied to a matching directory under "libJar". You may specify an explicit filename or use wildcards such as "*.*" (e.g. "*.png").
install	-Optional- This attribute is used during provisioning to strip optional directories for headless devices. Valid values are "runtime", "ui", and "doc". The default is defined by module element.

Exercse: Building and Testing the Module

1. Double click on the build.xml file under the myFirstComponent Java Project in the Package Explorer. The build XML file will display in the Eclipse editor.
2. Modify the build.xml file to include the com.examples package which should contain the java source code file BFirstComponent.

```
<module
  name = "myFirstComponent"
  bajaVersion = "0"
  preferredSymbol = "mfc"
  description = "A module containing a simple BComponent"
  vendor = "Tridium"
>

  <!-- Dependencies -->
  <dependency name="baja" vendor="Tridium" vendorVersion="3.6" />

  <!-- Packages -->
  <package name="com.examples" />

</module>
```

Warning

A common mistake when building a new module is to forget to include package and dependency declarations in the build.xml file. If your module is not building correctly, check the build.xml first.

3. From the Eclipse Niagara Plugin menu, select **Niagara Tools -> Build Full** and check the Eclipse console for a successful compilation. The module will be copied to the modules directory of the Niagara installation selected as the Niagara AX Home path of the Niagara Plugin.

4. Open the Niagara Workbench from the Niagara Console window or through Start menu of your platform.

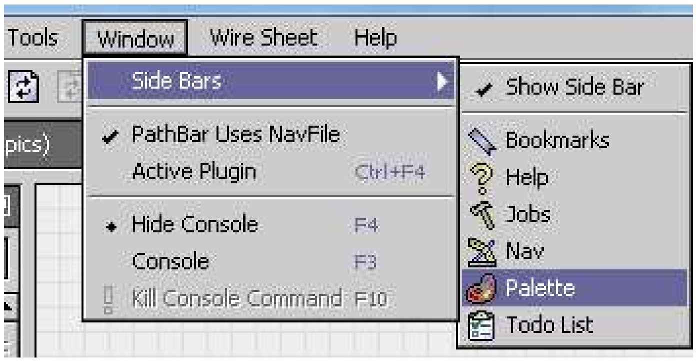
5. If you do not have a station, you can create a station using the New Station Wizard available in the Workbench toolbar. Select Tools -> New Station and follow the instructions of the wizard to create a new station instance.

   Once you have a station available, open a Niagara Console window and start the station using the station <station name> command syntax.

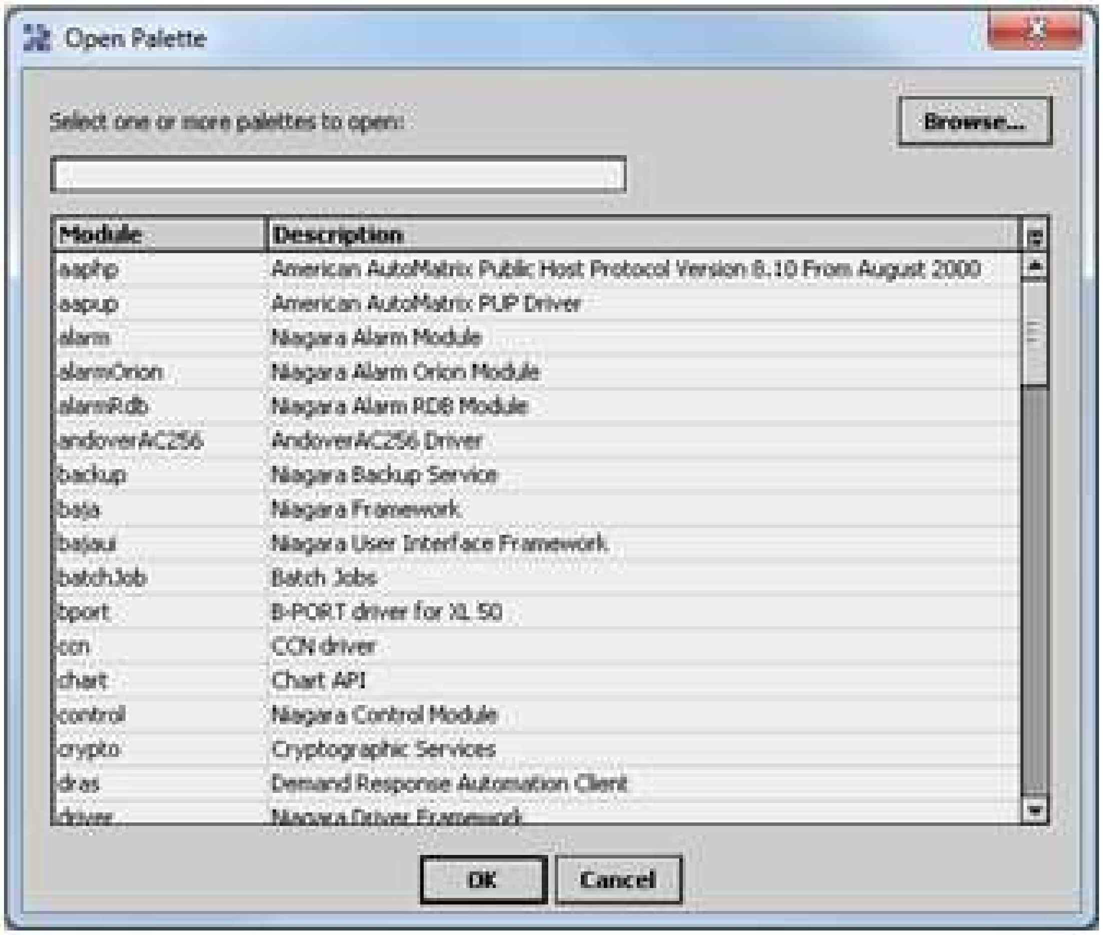
6. Connect to your running station instance using the Workbench. This can be done by clicking the Folder icon in the Workbench toolbar and selecting the Open Station (fox) command. Set the IP address of the running station to localhost and connect to the station.

7. If the Palette side bar is not already visible, display the sidebar by selecting Window -> Side Bars -> Palette from the Workbench menu bar.

Workbench Sidebar Menu



8. In the Palette sidebar, click the Open Palette command icon, which is a folder icon. The Open Palette dialog will display.



Select one or more palettes to open:

Module	Description
aaphp	American AutoMatrix Public Host Protocol Version 8.10 From August 2000
aapup	American AutoMatrix PUP Driver
alarm	Niagara Alarm Module
alarmOrion	Niagara Alarm Orion Module
alarmRdb	Niagara Alarm RDB Module
andoverAC256	AndoverAC256 Driver
backup	Niagara Backup Service
baja	Niagara Framework
bajaui	Niagara User Interface Framework
batchJob	Batch Jobs
bport	B-PORT driver for XL 50
ccn	CCN driver
chart	Chart API
control	Niagara Control Module
crypto	Cryptographic Services
dras	Demand Response Automation Client
driver	Niagara Driver Framework

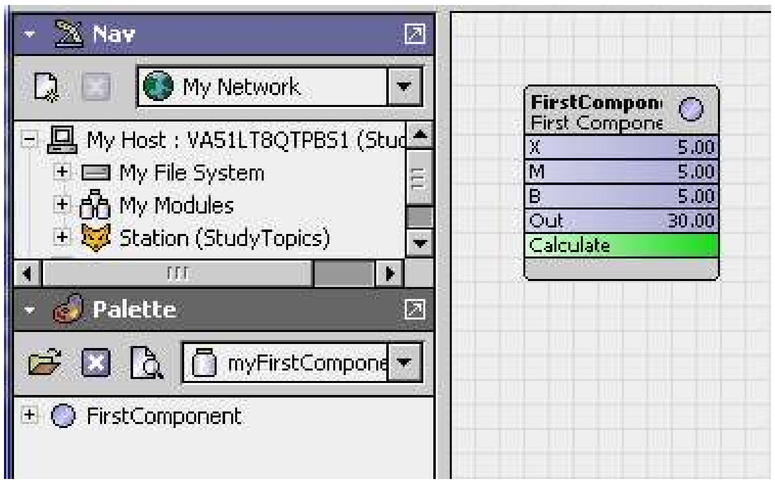
9. In the text field, type myFirstComponent and the myFirstComponent module should be listed among the available modules to select. Select this module in the list and click OK.

10. In the Palette sidebar, verify that an icon displays next to the text FirstComponent. This is a draggable instance of our BComponent.

11. Create a folder under the Config nav tree of the running station. Double click the folder. The Wiresheet View of the folder should display.

12. Drag and drop the FirstComponent instance from the palette into the running station.

Wiresheet view of the module and component



13. Modify the m, x, and b properties through the property sheet view. Right click on the component in the wiresheet view or the property sheet view and select Actions -> Calculate. Verify that as you change the values of m, x, and b, the out property value changes accordingly.

Summary

Modules are JAR files that include compiled java class files, packages, and resources that address domain specific functionality. Modules include descriptor files that make each module pluggable into the Niagara framework.

The New Module Wizard provides an easy way to create module files and directories and can be accessed directly from within the Eclipse environment.

BObjects provide Type information which is used to uniquely identify objects across the collection of modules that is the Niagara Framework. BComponents extend the BObject model and include Slots to define the Properties, Actions, and Topics of a component. These Property, Action, and Topic slots are the basis of Niagara's Component Oriented Programming model.

More information about Modules, Slots, and the BObject model can be found in the Developer Guide of the help documentation available both through the Niagara Plugin and through Workbench.

Solutions

To download individual NiagaraAX-(3.4 through 3.7) version source files click the "Files" tab at the top of the page.

Product Version

Description (cont.)