

# SinaXe Users Guide

Joseph Coffland  
jcofflan@users.sourceforge.net

September 15, 2006

**Copyright ©2004, Joseph E. Coffland** Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included with this documentation, [GNU Free Documentation License](#).

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>The Structure of a SinaXe Program</b>	<b>3</b>
<b>3</b>	<b>Some Examples</b>	<b>4</b>
3.1	Hello World . . . . .	4
3.2	A Useful Example: The List Editor . . . . .	5
3.2.1	The Menu . . . . .	5
3.2.2	The Frame . . . . .	5
3.2.3	The File Loader . . . . .	6
3.2.4	List, Text Field and Button Components . . . . .	6
3.2.5	Mapping the Components Onto XML Data . . . . .	7
3.2.6	The Layout . . . . .	8
3.2.7	Retargeting To A New XML Format . . . . .	8
<b>4</b>	<b>The SinaXe Markup Language (SML)</b>	<b>9</b>
4.1	Components . . . . .	9
4.1.1	Properties . . . . .	10
4.1.2	Input and Output Ports . . . . .	10
4.1.3	Queries and Subscriptions . . . . .	11
4.1.4	Variables . . . . .	11
4.2	Mappings . . . . .	11
4.3	Composites . . . . .	13
4.4	Layout . . . . .	13
<b>5</b>	<b>XPath and XUpdate</b>	<b>14</b>
<b>6</b>	<b>SinaXe Java Interface</b>	<b>14</b>
6.1	sinaxeInit . . . . .	14
6.2	sinaxeExit . . . . .	15
6.3	sinaxeGetComponent . . . . .	15
6.4	sinaxeGetDocumentation . . . . .	15
6.5	sinaxePointToContext . . . . .	16
6.6	SinaxeRuntime interface . . . . .	16
6.6.1	getFullName and getName . . . . .	17
6.6.2	exit . . . . .	17
6.6.3	getNeighborGraphic . . . . .	17
6.6.4	Variables . . . . .	17
6.6.5	Event Ports . . . . .	17
6.6.6	Queries and Subscriptions . . . . .	18
6.6.7	XPath function registration . . . . .	18
6.6.8	System errors and warnings . . . . .	18
6.7	SinaxeProperty interface . . . . .	18
<b>7</b>	<b>Web Site</b>	<b>19</b>

## 1 Introduction

New XML formats are emerging daily. Most formats could greatly benefit from an editor which provides a meaningful environment in which end users can interpret and manipulate the data. Typically, developers spend valuable time writing custom software to accomplish this task, settle for using one of the many tree based XML editors or just use a plain text editor.

SinaXe stands for SinaXe Is Not An XML Editor. SinaXe provides a set of graphical components which can be easily mapped on to arbitrary XML. Full featured XML editors are constructed in days instead of months and are easily modified to keep up with changing formats.

The rest of this document explains how to use the SinaXe language to map SinaXe components on to XML data.

## 2 The Structure of a SinaXe Program

This section provides a brief description of the parts of a SinaXe program. The sections that follow describe these concepts in more detail.

A SinaXe program is a network of components called a **composite**. The **component** is the most basic element. Components generally have a graphical user interface and a position in the composite layout, but not always. When SinaXe starts it first initializes all its components. At this point a component can read its **properties**, register **queries**, **subscriptions**, **variables** and **input and output ports** with the runtime system. However, no communication is allowed at initialization time.

After initialization components may receive events from input ports, subscriptions or input devices such as a mouse or keyboard. The component can react to these events by running registered queries, emitting events on output ports or possibly changing the value of a registered variable.

Mappings are used to connect components to other components and to the XML data they operate on and to form the external interfaces of composite components. There are several types of mappings. The most important mappings are between component input and output ports. These mappings are called **event maps**. Other kinds of mappings include **query maps**, **property maps** and **variable maps**. Query maps are used to map requests for data to the actual XML data the component will operate on. Property and variable maps are used to defer mapping to the parent composite component, thus forming the composite's external interface. Event and query maps can also add to the parent's interface.

Queries are mapped to XPath statements. When a component calls a query it provides a context for that query. The XPath statement will execute on this context to extract the query result. This enables components to read XML

data. Output ports can be mapped to XUpdate statements which modify the XML. Output events can also carry data which serves as the context for the XUpdate statement. These two languages, XPath and XUpdate, provide the glue between components and XML data and make it possible for components to operate on previously unknown data formats.

Composite components can also act just like regular components. Composites may have ports, properties, queries and variables just like regular components, but these are really mapped internally to its child components. Since every SinaXe program is simply a composite component it is possible to treat complex applications as simple components. You can make new applications by combining and mapping existing SinaXe applications.

To summarize, a SinaXe program is a composite component. Composites are collections of child components, maps and possibly other composites. Components can be configured with properties and can execute queries, send and receive events from ports and set the values of their variables. Components can be connected with maps. Maps can also map queries and output events on to XML data. Ports, properties, queries and variables can all be mapped to their parent composite component's external interface.

The next section contains a simple example of a SinaXe program which demonstrates some of these concepts.

## 3 Some Examples

What follows are several simple examples of SinaXe programs.

### 3.1 Hello World

```
<composite name="hello_world">
  <component name="frame" src="class:org.sinaxe.components.SinaxeEditorFrame">
    <property name="title" value="Hello World!"/>
    <property name="size">
      <property name="width" value="300"/>
      <property name="height" value="100"/>
    </property>
  </component>

  <map source="frame.closing" dest="frame.exit"/>
  <map source="init" dest="frame.show"/>
</composite>
```

The above example will open a window with the title "Hello World!". The property `title` is used to configure this title. The `size` property is used to set the frame's dimensions. Each particular component may have different properties, ports, queries and variables. These should be described in the components documentation.

The example maps the event `init` to the component's input port `show`. `init` is a special event generated by the SinaXe runtime which starts the program. It also maps the frame's `closing` event to `frame.exit`. This mapping causes the program to end when the frame closes.

This example shows a very simple SinaXe program, but it only demonstrates a small part of the picture. We aren't manipulating any XML data yet. The example in the next section demonstrates more of SinaXe's features.

## 3.2 A Useful Example: The List Editor

This section will walk you through creating a simple list editor in SinaXe. A list editor is not the most exciting example, but it remains simple while still demonstrating a complete SinaXe application which manipulates XML data. The complete code for this example can be found here: [list.sml](#).

### 3.2.1 The Menu

First off we create a menu with the following code.

```
<composite name="list_editor">

  <!-- Menu -->
  <component name="openitem" src="class:org.sinaxe.components.SinaxeMenuItem">
    <property name="text" value="Open"/>
  </component>
  <component name="saveitem" src="class:org.sinaxe.components.SinaxeMenuItem">
    <property name="text" value="Save"/>
  </component>
  <component name="exititem" src="class:org.sinaxe.components.SinaxeMenuItem">
    <property name="text" value="Exit"/>
  </component>

  <component name="filemenu" src="class:org.sinaxe.components.SinaxeMenu">
    <property name="text" value="File"/>
    <property name="menu">
      <property name="item" value="openitem"/>
      <property name="item" value="saveitem"/>
      <property name="separator"/>
      <property name="item" value="exititem"/>
    </property>
  </component>
</composite>
```

### 3.2.2 The Frame

Now we add a frame like in the previous example. Except here we add a few more properties.

```

<!-- Main Frame -->
<component name="frame" src="class:org.sinaxe.components.SinaxeEditorFrame">
  <property name="title" value="List Editor"/>
  <property name="size">
    <property name="width" value="400"/>
    <property name="height" value="400"/>
  </property>
  <property name="menubar">
    <property name="menu" value="filemenu"/>
  </property>
  <property name="child" value="layout"/>

  <map source="closing" dest="exit"/>
</component>
<map source="init" dest="frame.show"/>

```

The property `menubar` points to the previously created `filemenu` designating it as the frame's menu. The `child` property puts a child component in the frame, in this case the child is named `layout`. The `layout` component will come later.

### 3.2.3 The File Loader

Next we create a file loader which is able to read and write XML files.

```

<!-- File Loader -->
<component name="fileloader" src="class:org.sinaxe.components.SinaxeFileLoader">
  <property name="parent" value="frame"/>
</component>

<!-- Menu Actions -->
<map source="openitem.action" dest="fileloader.open"/>
<map source="saveitem.action" dest="fileloader.save"/>
<map source="exititem.action" dest="frame.exit"/>

```

The menu actions are mapped to the file loader's `open` and `save` input ports and the frame's `exit` port.

### 3.2.4 List, Text Field and Button Components

Next we create some components to put in the frame.

```

<!-- List -->
<component name="list" src="class:org.sinaxe.components.SinaxeList">
  <property name="subscribe"/>
</component>
<map source="fileloader.opened" dest="list.load"/>

```

```

<!-- Text Field -->
<component name="text" src="class:org.sinaxe.components.SinaxeTextField"/>
<map source="list.selectionchanged" dest="text.load"/>

<!-- Buttons -->
<component name="updatebutton" src="class:org.sinaxe.components.SinaxeButton">
  <property name="text" value="Update"/>
</component>
<component name="addbutton" src="class:org.sinaxe.components.SinaxeButton">
  <property name="text" value="Add"/>
</component>
<component name="delbutton" src="class:org.sinaxe.components.SinaxeButton">
  <property name="text" value="Delete"/>
</component>

```

Notice that the XML coming from the file loader is mapped to the list component. Also, the list's `selectionchanged` port is directed to the text field. This causes the text of the currently selected list element to be displayed in the text field.

### 3.2.5 Mapping the Components Onto XML Data

Now comes the interesting part. The components created in the previous section are mapped on to an XML data format.

```

<!-- XML Mappings -->
<map source="list.text">@value</map>
<map source="list.list">list/item</map>
<map source="text.text">@value</map>
<map source="updatebutton.action">
  <xupdate:update select='$list.selection/@value'>
    ${$text.value}
  </xupdate:update>
</map>
<map source="addbutton.action">
  <xupdate:append select='$list.context/list'>
    <xupdate:element name="item">
      <xupdate:attribute name="value">${$text.value}</xupdate:attribute>
    </xupdate:element>
  </xupdate:append>
</map>
<map source="delbutton.action">
  <xupdate:remove select='$list.selection'/>
</map>

```

To understand the above mappings it helps to understand the XML data they are designed to manipulate. Below is some possible input data. Note what follows is **not** part of the SinaXe program.

```
<list>
  <item value="A"/>
  <item value="B"/>
  <item value="C"/>
  <item value="D"/>
</list>
```

In the previous SML code, the list component's `list` query is mapped to the `item` element children of the `list` element with the XPath statement `list/item`. Also, the `text` queries of both the list and text field components are mapped to the `value` attribute of the `item` element. It is important to know that the list component uses the `item` elements as the context to its `text` query.

Finally the three buttons are mapped to XUpdate statements. The update button selects the `value` attribute of the currently selected `item` elements and changes their text to the value stored in the text field's `value` variable. The add button appends a new `item` element using the text in the text field's `value` variable. The delete button removes all the selected list elements.

### 3.2.6 The Layout

Only the layout of the components remains. We will use the `SinaxePackerLayout` component which uses a special layout language to place the components on the screen.

```
<!-- Layout -->
<component name="layout" src="class:org.sinaxe.components.SinaxePackerLayout">
  <property name="layout">
    <panel layout="fill=both">
      <panel layout="side=top;fill=x">
        <port component="text" layout="side=left;fill=x;expand=true"/>
        <port component="updatebutton" layout="side=left"/>
        <port component="addbutton" layout="side=left"/>
        <port component="delbutton" layout="side=left"/>
      </panel>
      <port component="list" layout="side=bottom;expand=true;fill=both"/>
    </panel>
  </property>
</component>
```

### 3.2.7 Retargeting To A New XML Format

It is very easy to retarget SinaXe programs to new XML data formats. From the example in this section all we need to replace is the code from the section

entitled Mapping the Components Onto XML Data(3.2.5). But, first lets come up with some new XML data.

```
<html>
<body>
  <ol>
    <li>1</li>
    <li>2</li>
    <li>3</li>
    <li>4</li>
  </ol>
</body>
</html>
```

The above example data is in XHTML format. Note that SinaXe cannot operate directly on HTML because it is not XML. However, it is very easy to convert HTML to XHTML with free tools like [tidy](#).

To edit this new format with our list editor we replace the XML mapping code in our example with the following:

```
<!-- XML Mappings -->
<map source="list.text">text()</map>
<map source="list.list">html/body/ol/li</map>
<map source="text.text">text()</map>
<map source="updatebutton.action">
  <xupdate:remove select='$list.selection/text()'/>
  <xupdate:append select='$list.selection'>
    <xupdate:text>${$text.value}</xupdate:text>
  </xupdate:append>
</map>
<map source="addbutton.action">
  <xupdate:append select='$list.context/html/body/ol'>
    <xupdate:element name="li">
      <xupdate:text>${$text.value}</xupdate:text>
    </xupdate:element>
  </xupdate:append>
</map>
<map source="delbutton.action">
  <xupdate:remove select='$list.selection'/>
</map>
```

## 4 The SinaXe Markup Language (SML)

### 4.1 Components

Components are the most basic element in SML. Components generally have a graphical user interface, but not always. When SinaXe starts it first initializes

all its components. At this point a component can read its properties, register queries, variables and input and output ports with the runtime system. However, no communication is allowed at initialization time. After initialization components may receive events either from their input ports or from an input device such as a mouse or keyboard. The component can react to this input by running a registered query, emitting an event on an output port or possibly changing the value of a registered variable.

A the `component` element is only required to have two attributes, `name` and `src`. The `name` attribute must be a unique name within the components parent `composite`. The `name` attribute is used to identify the component. The `src` attribute is used to load the actual component code. There are currently two forms supported. Either the `class:<class path>` form or the `sml:<sml file>` form. The first loads a Java class. The second loads an SinaXe program as a component.

The ports, properties, queries and variables of a particular component are dependent on that component. Consult the documentation or source code of the component of interest.

#### 4.1.1 Properties

Properties are used to configure components. A property consists of the `property` element, a `name` attribute and optionally a `value` attribute. Properties may also have child properties. Some properties are required other are optional. A few examples of valid properties are show below:

```
<property name="text" value="A"/>

<property name="clickable"/>

<property name="menu">
  <property name="item" value="openitem"/>
  <property name="item" value="saveitem"/>
  <property name="separator"/>
  <property name="item" value="exititem"/>
</property>
```

#### 4.1.2 Input and Output Ports

A component may have any number of input or output ports. They must however, have unique names within that component. The type of data a port is allowed to accept or emit is defined by the component. Possible data types are `string`, `number`, `XML node`, `list` and `null`. Any combination of these may be allowed. Valid SinaXe data is one of these types. It is not required that ports are mapped to anything. If they are not mapped they will be ignored.

### 4.1.3 Queries and Subscriptions

Components use queries to get information about XML data. The result of a query is always one the **string**, **number**, **XML node**, **list** or **null** data types. Some queries are required and must be mapped to avoid an error at program initialization. When a component executes a query it provides a context. This context may be **null**. If mapped a query must map to an XPath statement.

Components may choose to create subscriptions from queries. A subscription is always associated with the query from which it was derived. Once a component has created a subscription it can run the query on that subscription with a particular context. The results of the query are returned as normal. From that point on if the XML data changes in any way that would change the results of the query, the component will be automatically notified of such changes. The subscription will remain active until the component either releases the subscription or reruns the query with a different context.

The benefit of subscriptions is that components automatically update their view of the data when it is changed. The down side is that if a subscription change event causes the component to fire new events a chain reaction of possibly unnecessary events can result. For this reason subscriptions should generally not actuate new events. Many components do not automatically subscribe to their data unless they are configured with the **subscribe** property. This is merely a convention and not every component follows it.

### 4.1.4 Variables

Components may define variables. The component can then set the value of this variable. Variables can provide extra context information to queries and updates. A component's variables can be referenced with the following syntax: **\$component.variable**. Or simply as **\$variable** within the components definition. However, a component cannot reference variables in another component from within its own definition. In other words the following is not allowed:

```
<component name="A" src="class:..." />
<component name="B" src="class:..." >
  <map source="aquery">$A.value</map>
</component>
```

However, the following is valid:

```
<component name="A" src="class:..." />
<component name="B" src="class:..." />
<map source="B.aquery">$A.value</map>
```

## 4.2 Mappings

There are several types of mapping possible in a SinaXe program. These include **event mapping**, **property mapping**, **variable mapping** and **query mapping**.

Event mappings are used for three distinct purposes. To map events from one component's output port to another components input port, to map an input or an output port from a component to its parent's interface, or to map an output port to an XUpdate statement. Examples of these three uses are given below:

```
<composite name="parent">
  <component name="A" src="class:..."/>
  <component name="B" src="class:..."/>

  <!-- Connecting components -->
  <map source="A.out" dest="B.in"/>

  <!-- Mapping to the parent interface -->
  <map source="input" dest="A.in"/>
  <map source="B.out" dest="output"/>

  <!-- Mapping to XUpdate -->
  <map source="A.update">
    <xupdate:remove select='.'/>
  </map>
</composite>
```

Property and variables mappings are used only to map properties and variables up to the parent component. An example follows:

```
<composite name="parent">
  <component name="A" src="class:..."/>

  <map source="A.aproperty" property="aproperty"/>

  <map source="value" variable="A.value"/>
</composite>
```

Notice that variable mappings run in the opposite direction as property mappings. This is because properties are requested from the component whereas variables are queried from outside the composite.

Query mappings take two forms. A query may either be mapped to its parent's interface or directly to an XPath statement. A components query mapping can also occur inside the components definition or out. Examples follow:

```
<composite name="parent">
  <component name="A" src="class:...">
    <!-- A mapping to an XPath statement from with in the component definition. -->
    <map source="aquery">selects/some/nodes</map>
  </component>
```

```

<!-- An XPath mapping outside the component definition -->
<map source="A.anotherquery">select/some/@attribute</map>

<!-- A query mapping to the parent -->
<map source="A.yetanotherquery" query="parentquery"/>
</composite>

```

To summarize map elements always have a `source` attribute and may have one of the `dest`, `property`, `variable` or `query` attributes or none. If the second attribute is not present the mapping is either to an XPath in the case of a query mapping or an XUpdate in the case of an out port mapping. Any other combinations are invalid.

### 4.3 Composites

Composites are components composed of child components and mappings. A composite can be treated as a regular component when it is loaded as one via the `src="sml:<file name>"` or when it is the child of another composite. These two cases are illustrated below.

```

<composite name="parent">
  <component name="ReallyAComposite" src="sml:AComposite.sml"/>

  <composite name="AChildComposite">
    <component name="A" src="class:..." />
    <component name="B" src="sml:..." />
  </composite>
</composite>

```

### 4.4 Layout

In order to place SinaXe components on the screen there must be some sort of layout. Certain components are allowed to refer to the graphical interface of neighboring components. This allows components such as `SinaxeEditorFrame` to place the component referred to in its `child` property in its graphical representation. This sort of graphical reference is the simplest form of layout. Not all components have a graphical representation. An error will occur if such a component is referenced by a component which requires a graphical component reference. Since composite components may also be treated as regular components they must also be able to have a graphical representation. The convention used in SinaXe is that the last component of a composite contains its graphical representation. For that purpose a layout component is often the last component of a composite.

Special layout components are provided to allow more complex layouts. These components are able to read a special `layout` property which is allowed

to contain `panel` and `port` elements. The top most element of the `layout` property must either be a single port or a panel. Panels can contain ports and other panels. Ports refer to components in the current composite component. A component can only appear in one place in the layout. The `component` attribute of the `port` element specifies which component shall be displayed in that port. Both ports and panels have a `layout` property. The value of this property is dependent on the specific layout component in use, but it is used to configure the layout. Currently there are two layout components; `SinaxePackerLayout` and `SinaxeGridLayout`.

## 5 XPath and XUpdate

The XPath and XUpdate languages are used to map SinaXe components on to XML data. Admittedly, usage of XPath is stretched beyond its intentions in SinaXe and XUpdate is limited, verbose and no longer supported by its creators. What SinaXe really needs is a special purpose XML processing language. Good fortune would have it that just such a language is underway. See [XmlPL](#), the XML Processing Language. XmlPL is still in its early stages so it will be awhile until it is integrated into SinaXe.

## 6 SinaXe Java Interface

This section describes the interface between SinaXe graphical components written in the Java Swing environment and the SinaXe run-time system. It is assumed the reader has some knowledge of the Java programming environment.

SinaXe GUI components are implemented as Java classes which implement the `SinaxeComponent` interface. All SinaXe components must implement this interface. In the current implementation of SinaXe most of the `SinaxeComponent`s are merely wrappers for Java Swing components and therefore extend a Swing component directly. The `SinaxeComponent` interface is listed below.

```
public interface SinaxeComponent {
    public void sinaxeInit(SinaxeRuntime runtime, SinaxeProperty properties)
        throws SinaxeException;
    public void sinaxeExit() throws SinaxeException;
    public Component sinaxeGetComponent() throws SinaxeException;
    public String sinaxeGetDocumentation();
    public Object sinaxePointToContext(Point source) throws SinaxeException;
}
```

### 6.1 `sinaxeInit`

The `sinaxeInit` function is called on every SinaXe component instance at program startup beginning from the root of the SinaXe XML tree and proceeding in document order. At the time a component is initialized there is no guarantee

that the other components in the system have been initialized. For this reason SinaXe components are not allowed to send events or execute queries during the call to `sinaxeInit`. This ensures that no component receives an event before has been initialized.

The call to `sinaxeInit` passes two parameters, the `SinaxeRuntime` and the `SinaxeProperty` tree. The component must register its event input and output ports, variables, and queries with the `SinaxeRuntime` object during the call to `sinaxeInit`. In addition the component should examine its properties and configure itself accordingly. The `SinaxeProperty` structure is an object based view of the XML properties specified in the SinaXe markup.

Since some properties, ports, variables and queries are common to a group of components or even all components, some general purpose initialization functions are provided as static methods of the `SinaxeUtil` class. These default initializations are performed by calling a `SinaxeUtil.loadDefaultInits` method. For example, mouse events which may be generated by any GUI component are registered here.

Both the `SinaxeRuntime` and `SinaxeProperty` objects are described in more detail below.

## 6.2 `sinaxeExit`

This function is called for each `SinaxeComponent` upon program termination. This is where `SinaxeComponents` should do any necessary cleanup, such as releasing any system resources they may have acquired at run-time.

## 6.3 `sinaxeGetComponent`

This function make it possible for components to access to the Java Component interface of another `SinaxeComponent`. This is necessary in Java for Java Swing containers which need to add their children. Not all `SinaxeComponents` have a graphical representation; in which case they may return null in this function.

To access the Java Component interface of another `SinaxeComponent` a call to `SinaxeRuntime.getNeighborGraphic` is necessary. Given the name of the neighboring `SinaxeComponent` this will return the Component interface, null, or generate a run-time error if the `SinaxeComponent` is not found. The path to the neighboring component is specified by the SinaXe markup via a property.

## 6.4 `sinaxeGetDocumentation`

This function should return a string of text documenting the ports, queries, variables and properties of the `SinaxeComponent`. As of this writing this function is not used.

## 6.5 `sinaxePointToContext`

The `sinaxePointToContext` function is used to turn an (X,Y) coordinate into a SinaXe XML context. This is not relevant for most `SinaxeComponents` and in most cases just returns null. For components which display XML data for example as a tree, graph or list, this function is used to get a pointer to the data represented by the tree or graph node or list entry given a coordinate. For example, if the user clicks on a list node this generates a mouse clicked event which contains the (X,Y) position of the click. A call to `sinaxePointToContext` can be used to access the XML data represented by the clicked list entry.

## 6.6 `SinaxeRuntime` interface

The `SinaxeRuntime` interface is used by `SinaxeComponents` during their `sinaxeInit` call to register event ports, queries and variables among other things. The interface is listed below.

```
public interface SinaxeRuntime extends SinaxeDataTypes {
    public String getFullName();
    public String getName();

    public void exit(int exitCode);
    public Component getNeighborGraphic(String name, RuntimeComponentBase context) throws SinaxeException;
    public Component getNeighborGraphic(SinaxeProperty prop) throws SinaxeException;

    public SinaxeVariable lookupVariable(String name) throws SinaxeException;
    public SinaxeVariable registerVariable(String name) throws SinaxeException;
    public SinaxeInPort getInPort(String name) throws SinaxeException;
    public SinaxeOutPort getOutPort(String name) throws SinaxeException;
    public SinaxeInPort registerInPort(String name, int datatype) throws SinaxeException;
    public SinaxeOutPort registerOutPort(String name, int datatype) throws SinaxeException;
    public SinaxeQuery getQuery(String port) throws SinaxeException;
    public SinaxeQuery getQuery(String port, boolean required) throws SinaxeException;
    public SinaxeSubscription getSubscription(String port) throws SinaxeException;
    public SinaxeSubscription getSubscription(String port, boolean required) throws SinaxeException;
    public void registerFunction(SinaxeFunction function) throws SinaxeException;

    public void issueWarning(String warning, Exception cause);
    public void issueWarning(String warning);
    public void issueError(String error, Exception cause);
    public void issueError(String error);
    public void issueFatalError(String error, Exception cause);
    public void issueFatalError(String error);
}
```

### 6.6.1 `getFullName` and `getName`

These functions give the `SinaxeComponent` access to its name as specified in the SinaXe markup. Every `SinaxeComponent` instance has a unique full name and a unique name with in its XML context. The full name is a '.' separated string of names starting from the root SinaXe node, continuing down the component's ancestors and ending with the component's own local name.

### 6.6.2 `exit`

A call to the `SinaxeRuntime.exit` function will terminate the SinaXe program.

### 6.6.3 `getNeighborGraphic`

The `getNeighborGraphic` function is used to gain access to the Java GUI Component interface for another `SinaxeComponent` instance in the current program. This function finds the neighboring component given a name path and return the result of calling `SinaxeComponent.sinaxeGetComponent` on that component.

### 6.6.4 Variables

The `registerVariable` function is used to tell the runtime system about a `SinaxeComponent`'s SinaXe variables. This makes registered variables accessible from SinaXe markup. If a variable is registered more than once or looked up before it is registered a run-time error will be generated.

### 6.6.5 Event Ports

Registering event ports provides the `SinaxeComponent` with a handle which it can use to send and receive events. Port names must be unique with in a `SinaxeComponent`. Restrictions on the allowable datatypes that may be used with the port may be specified with the `datatype` parameter during registration. Possible datatypes are specified in the `SinaxeDataTypes` class.

Events can be sent over output ports using the `SinaxeOutPort` handle directly. To receive events from an input port the `SinaxeComponent` must register a `SinaxePortListener` with the `SinaxeInPort` handle. For example, to register an input port with the name "in", which can only receive a number or a string you would write the following:

```
inPort = runtime.registerInPort("in", DT_NUMBER | DT_STRING);
inPort.addListener(new SinaxePortListener() {
    public void sinaxePortEvent(SinaxeInPort port, Object data) {
        if (data instanceof String) {
            // Do something
        } else if (data instanceof Number) {
            // Do something else
        }
    }
})
```

```
});
```

### 6.6.6 Queries and Subscriptions

Queries and subscriptions are registered much like event ports. In both cases the registered name must be unique within the `SinaxeComponent`. The `required` parameter specifies whether the `SinaxeComponent` must be provided with the registered query or subscription. The default is that it is not required. If the query or subscription is required but is not specified in the SinaXe markup then an error will be generated at run-time.

Once a subscription is registered a `SinaxeSubscriptionListener` callback must be registered with the subscription handle. Then, before any callback will be received the subscription must be initialized by running a query on the subscription. This will cause the subscription to subscribe to the relevant XML data points which will cause callbacks to be called when the query result has changed. Subscriptions should be used with care as they can cause unexpected chainreactions within the SinaXe program and result in degraded performance.

### 6.6.7 XPath function registration

`registerFunction` is used to add XPath functions which can be called from SinaXe markup. Function names must be unique. `registerFunction` is generally not called from a `SinaxeComponent`.

### 6.6.8 System errors and warnings

SinaXe components may generate errors and warnings in one of two ways. Either by throwing a `SinaxeException` or by issuing a call to one of the `SinaxeRuntime` error and warning functions. Warnings and errors are simply printed to the standard error stream. Fatal errors cause program termination.

## 6.7 SinaxeProperty interface

The `SinaxeProperty` interface is used to gain access to `SinaxeComponent` properties specified in SinaXe markup. These properties are organized as a tree. In other words, properties may have child properties. A root property is automatically added to the top of the tree. The root property is only a container for the component's top level properties.

Property names do not have to be unique. The `get` function will return a `SinaxeProperty` or null if it does not exist. The `require` function will generate an error if the specified property does not exist. The `has` function returns true if the named property exists. `getIterator` can be used to get either a list of all the child properties or only those matching the specified name.

## 7 Web Site

The SinaXe website can be found at <http://sinaxe.org>.

## 8 Software License

The SinaXe software is available under the terms of the [GNU Public License \(GPL\)](#). If you require a different license please contact [jcofflan@users.sourceforge.net](mailto:jcofflan@users.sourceforge.net) to discuss commercial options.