MPages Component Standard

Version 1.1

Dec 19th, 2012

Contributors: UW Medicine, University of New Mexico, University of Maryland, Virginia Mason, Boston Children's, Seattle Children's, Trinity, Virginia Commonwealth University, Cerner, Mayo Jacksonville, St. Jude's, Moffitt Cancer Center.

Contents

1. Goals

1.1 Goal

Allow a user to develop a component without reference to a specific framework implementation. An MPage developer should be able to use any of those components on a single MPage in an interoperable fashion.

1.2 Community Defined Open Standard

The component standard will be developed, defined and supported by a community of participants including Cerner clients, Cerner engineers and others in support of the goal listed above. The standard will be open source allowing anyone to develop a solution that supports the standard (subject to client MPages licensing restrictions and requirements).

1.3 Test Case

1.3.1 Develop one of more framework implementations of the standard with different base-class implementations. A component built to work with one framework should work seamlessly with another.

1.3.2 Develop a component and validate the build without a framework implementation. The component should be easily used by an existing framework.

1.3.3 Deploy a standard compliant component and use in the Cerner MyPages.

2. Definitions

2.1 Base-Class. An object-oriented programming construct that defines basic functionality that another class will inherit from (i.e., a car class would be a base-class for a Toyota class).

2.2 (a) Base-Class. In this document, base-class is generally used to refer to the MPage.Component base-class that all components inherit from. This is specific to each framework and the internal implementation may vary from page to page; however, the functionality exposed to the Component will be consistent based upon this standard. For most purposes, the base-class can be considered part of an overall framework implementation.

2.3 Class. An object-oriented programming construct. A general concept of an instance object (i.e., a car is a class whereas my car is an object).

2.4 Component. In this document a component refers specifically to a component or widget that will be used on a MPage to display a particular set of data. For example, an allergy widget. It is possible to develop components for many purposes and use them on your MPages; however, the standard is defining the component standard specifically for those components that make up a summary MPage. This is commonly seen with Cerner summary MPages and other clients MPages where the page is composed of many components each displaying their specific data.

2.5 Component Developer. A component developer builds a component class that inherits from the MPage.Component base-class. To be standard compliant they must do so according to the specifications in this document. A component is built to the general standard, not to the specifications of a specific framework.

2.6 Framework. A framework is a specific implementation of the base-class and a set of tools used by an MPage “Page” designer. This term is often used interchangeably with the term Page because it determines how the “Page” will be built. There are multiple possible framework implementations that may have different internal implementations and different page facility functionality. However, the functionality exposed to the component developer must conform to this specification.

2.7 Framework Developer. A framework developer builds the framework logic that allow for an MPage designer to easily implement an MPage consisting of multiple components.

2.8 Interface. An interface is a well-defined method for interaction. In this case, it generally refers to the component and framework interacting through a method, property, event or other mechanism. This is the heart of the standard as it defines how the component and the framework/page interact.

2.9 Object. An object-oriented programming construct. An object is a specific instance of a class (i.e., a car is a class whereas my car is an object).

2.10 MPage/Page. An MPage is a specific page built, often using a framework and components to make implementation easier and quicker.

2.11 MPage/Page Developer.

An MPage developer uses a framework to implement an MPage built out of one or more components.

3. Component Architecture

3.1 JavaScript-Based Components

Components will be browser-side JavaScript components that generate content and functionality in the browser rendering the MPage. JavaScript is the core technology in this architecture. This is opposed to back-end components which generate content using CCL and do not require a JavaScript implementation. This approach decouples the display and related functionality from the data source and related business logic and provides for more flexible functionality.

3.2 Class-Based Components

Components will be class-based implementations in JavaScript. This will allow for re-use of the component in multiple MPages as well as on the same MPage following object-oriented design principles.

3.3 Base-Class Inheritance

Components will inherit interfaces and functionality from a base-class supported by the framework. This base-class must be extended using JavaScript prototypal inheritance. A reference base-class in included with this standard; however, multiple base-classes are allowed under this standard, as long as each conforms to the requirements of the standard.

4. Scope

4.1 Black-box Design Principle

The component standard is architected around a black-box design principle. Any component specific functionality should be scoped within that component and should not interact with the page or other components except through the interfaces defined in this standards document.

In this principle, the concept of the black-box applies to both the component and the page/framework. From the component's perspective, the page should be a black-box which it interacts with only via the defined interfaces. From the page's/framework's perspective, the same is true. The page should never need to know how the component is implemented or what it is doing, except where exposed through the interfaces defined in this document. This is further illustrated in a diagram in Appendix 3.

4.2 Component to MPage Interaction

Components must only interact with the MPage in the following ways:

(a) The interfaces defined in this document, implemented as part of the component base-class
(b) Updates to the page DOM only within the appropriate scope as defined in this standards document

The framework cannot expose, and the component developer cannot access any method or property that is not explicitly supported by the standard.

4.3 Framework

Through-out this document, the concept of a framework will be referred to. The framework is ANY implementation of standard page-level functionality designed to support the base-class and the required functionality. It would also consist of any functionality used by an MPage developer to configure and build an actual MPage using components. Although a framework could theoretically use any base-class in its implementation, in general, a base-class and a framework are developed as one implementation that work seamlessly together to expose the interfaces required by this standard.

4.4 Rendering Page Scope

4.4.1 Rendering Page Scope

Each component will be scoped to the component's main Document Object Model (DOM) element. This is likely to be a DIV element. When the component is rendering content within the page (or attaching events, etc.) this will be confined to that element on the page only. This implies that the DOM element must exist on the page prior to component rendering.

4.4.2 Rendering Page Scope Exception (Tooltips and Popups)

Due to how Internet Explorer renders z-index positioned layers based upon their location in the DOM hierarchy, there is one exception to the page scope rendering rules. A component MAY create DOM elements as children of the BODY element (only) for the purposes of creating tooltips and popup layers. These elements must be rendered with an absolute position and z-index greater than 0 (1,000+ is recommended) to force them to display as floating layers above other component and page content.

4.4.3 Rendering Width

In order to avoid components accidentally affecting page level display, including the page layout, the width of a component must fit within the width of the target element. To make this work correctly, the component developer should either use percentage based sizing or check the width of the component container prior to rendering and force their component to fit using an absolute CSS width. If the component cannot render within that space, the overflow must either (a) be hidden or (b) scroll.

To maintain consistent rendering when the page resizes, each component must account for the resize event method by either (a) redrawing the component or (b) adjusting the absolute width.

Note: Version 2.0 of the component standard may define min and max width properties.

4.4.4 Rendering Height

The component's height may or may not be restricted by the framework. If the framework wishes to restrict the component's height the 'maxHeight' property will be set to the maximum number of pixels the component's DOM element may take up before the framework will start to apply scrolling. The framework also has the ability to set this property to 'null' which will signify that the component's height is unrestricted. The component can choose to abide by the 'maxHeight' property and ensure its content will fit inside that allotted space to prevent scrolling or it can ignore it and allow the framework to handle the scrolling automatically.

4.4.5 Component vs. Framework/Page Functionality

The following functionality is scoped to the page and will be rendered and managed by the MPage NOT the component:

- Any common component header that would be expected to contain the component title or other information and would exist directly above the component's main DOM element.
- Any component title and/or sub-title as rendered in the component header.
- Any header linking to a specific PowerChart tab. The page and framework will support any standard linking PowerChart as configured for that page if it is exposed within the header.
- Any header show/hide functionality. The page/framework is responsible for adding controls to allow a user to show or hide the whole component. This will support a more standard look and feel.
- Any standard page level functionality that manages the size of the component's main DOM element (e.g. overflow, more/less toggles, etc.). Individual components can implement their own ability to overflow and hide particular elements, but the overall ability to hide/show or overflow the main component div should be managed by the page to maintain consistent functionality. If the component needs to know the size of the DOM element for internal sizing, it can access the property directly from the DOM. This is standard compliant as each component is supposed to interact with its defined element space on the page. In the case of a page resize, the framework will identify the event as fire the event (which the component can listen to).
- Any content injected into the header should be treated as plain text. Injecting HTML code should be avoided as it blurs the distinction between the page's and component's scope.
- Any DOM content or functionality not specifically scoped to within the component's main DOM element

This design allows for consistent page level functionality and look and feel across components developed by multiple organizations. If the component interacts with the header, it will be expected to do so through the interfaces supported by the base-class implementation and defined in this document. The component will not directly write to the header's DOM element.

4.5 Component to Component Interaction

In the case that components need to interact with each-other, either to sync common events or to share common data, this must follow the black-box principle.

4.5.1 Common Page Scope (NOT ALLOWED)

Embedding common events or content on the page to share component scope is expressly NOT allowed. For example, a component cannot be defined to render or bind any content or event at the page level and by extension should NOT be aware of other components on the page.

4.5.2 Interfacing Mechanism

If components on a page interact with each other outside of the supported base-class interfaces they should do so using a common JavaScript mechanism which must:

(a) Be JavaScript based (versus DOM based)
(b) Conform to the JavaScript namespace conventions required in this standard
(c) Only interface with components that are specifically designed to interact with this mechanism
(d) NOT interact directly with any object outside of the Organization namespace or the component's DOM element.

As an example, this could consist of a common JavaScript collection object that each component would register with to exchange events and information.

By following the namespace scope conventions and only interacting with the components designed to work with it, this mechanism for component interfacing would be considered part of the component black-box architecture.

The implication here is that components that work together as a group must be explicitly designed to do so under a common namespace mnemonic. This will help insure proper documentation of the requirements as well as safe interaction and functionality.

4.6 Additional Public Interfaces

IT IS ABSOLUTELY PROHIBITED TO DEVELOP ADDITIONAL PUBLIC INTERFACES FROM A COMPONENT THAT WILL BE CONSUMED BY A USER OF THE COMPONENT. This is because any additional interface will potentially not be used in some frameworks and can cause interoperability issues. The most common example of these types of public interfaces would be a method making it easier to set component properties or generate a component. For example, a generate() method attached to the component allowing a page developer to generate a component with a single call.

INSTEAD OF THE COMPONENT DEVELOPER ADDING ADDITION PUBLIC INTERFACES FOR EASE OF USE, THIS SHOULD ALWAYS BE DONE BY THE FRAMEWORK DEVELOPER. The framework is designed to make it easier to load and manage components on a page. As such, this is directly within the scope of the framework's intended functionality. In addition, a framework is allowed to add any such interfaces without violating the standard or risking interoperability issues. For instance, if framework A exposes method Generate(), but framework B does not, this is not an issue. Because this is exposed at the framework level, it simply means that a developer working to build a page in CCL can use a single call in framework A, but may have to use multiple calls to load a component in framework B. In either case, any component can interoperably and interchangeably work in either framework (one is simply easier to use).

Additional component private functions, variables and methods (as well as client mnemonic private functions) are supported within the standard. It is only additional public interfaces that are not allowed.

5. JavaScript Namespace Scope

5.1 Namespace Management

All implementations of components or frameworks are expected to follow the namespace convention outlined below. The purpose of this convention is to predictably manage the scope of JavaScript objects, variables and functions and promote interoperability.

5.2 MPage

MPage is the base namespace object that must be implemented by any framework. This should be instantiated as a single object that will be used to contain all other page level functionality. Because the framework owns the MPage namespace, only a single framework can be used on any page.

Example(s)

```
var MPage = new Object();
```

5.3 MPage.Component

MPage.Component is the base-class component definition. This is the class that all components will inherit from. This is required as part of the standard inheritance model. A reference implementation of this class is provided with this specification; however, any framework can and general will implement its own base-class under the namespace MPage.Component. For a given framework and page, there can only be one base-class.

Example(s)

```
MPage.Component = function(){};
Mpage.Component.prototype.render = function(){};
```

5.4 MPage.[object/function/variable]

All custom functionality developed to support the framework should be scoped to the MPage object. This would include common data stores such as global property hashes, component references, private functions, etc. The framework should never expose global variables for use or accidentally pollute the global namespace. All namespaces must be scoped under below the MPage object.

Example(s)

```
MPage._compArray  = new Object();
MPage._getProperty = function(name){};
MPage.createPage = function(){};
MPage.CompArray[0] = new mnemonic.MyComponent;
```

5.5 [mnemonic]

All components must be created under a master namespace for each organization. This should generally consist of the organization's mnemonic. The mnemonic would be created as a singular object to scope the namespace of any component. It is generally a good idea to either have a master file that defines that mnemonic object, or check for its existing at run-time with each component or JavaScript file.

Example(s)

```
If (univwa == undefined){
        var univwa = new Object();
}
```

5.6 [mnemonic].[Component]

Each component would then be scoped to that namespace during creation and defined as an inherited child class of the MPage.Component base-class.

Example(s)

```
univwa.MyComponent = function(){};
univwa.MyComponent.prototype = new MPage.Component();
univwa.MyComponent.prototype.constructor = MPage.Component;
univwa.MyComponent.prototype.base = MPage.Component.prototype;
```

5.7 [mnemonic].[object/function/variable]

In the case that an organization implements common JavaScript functionality that crosses the components, this functionality must be scoped to the client's mnemonic master namespace. This would be used if there are common functions, global (to that organization) variables, collection objects, etc.

Example(s)

```
univwa.hideOverflowText = function(){};
univwa.sPowerChartName = "ORCA";
univwa.MyComponentCollection = new Object();
```

6. Base-Class Implementation

6.1 Support for Multiple Implementations

Each MPage implementation utilizing components (either as a single MPage or as part of a framework), must implement a base-class that meets the requirements outlined below. This can be the reference base-class published by the Component Committee, or it can be a custom base-class designed to implement functionality supported within that framework.

6.2 Conform to Component Interface Standard

The base-class will conform to the interface standards as required in this document.

6.3 Conform to Component Functional Standards

The base-class will conform to functional standards as required in this document, in addition to the specific interface definitions.

6.4 Base-Class Inheritance Support

The base-class will support standard JavaScript prototype inheritance without requiring any additional JavaScript libraries.

Example(s)

```
MPage.Component = function(){};
Mpage.Component.prototype.render = function(){ //base-class functionality };

univwa.MyComponent = function(){};
univwa.MyComponent.prototype = new MPage.Component();
univwa.MyComponent.prototype.constructor = MPage.Component;
univwa.MyComponent.prototype.base = MPage.Component.prototype;
univwa.MyComponent.prototype.render = function(){ //override };
```

6.5 Component Inheritance Mechanism

Every component will inherit from a base-class common to the page. However, the specific inheritance mechanism may vary from component to component. Although, each component may always inherit from the base-class using the standard JavaScript mechanism, alternatives may be supported. For example, a component that uses the MooTools library may use the new Class() pattern. This would require MooTools to be listed as a component dependency for that page. Regardless of the mechanism used, the resulting object must include the base class's prototype in its prototype chain.

6.6 Base-Class Name

The name of the base-class in any specific implementation will always be "**MPage.Component**".

6.7 Base-Class Supported Version

The base-class will provide a read-only public variable that will specify the standard version it is compatible with. This value will be set by the framework to identify. This can be used by the component developer to identify which functionality should be included or excluded as necessary to support reverse compatibility.

6.8 Component Minimum Required Version

The base-class will expose a public variable that will specify the standard version that the component developer requires. The framework can check the component's minimum required version to identify if it is compatible with this framework and should be loaded. The default value assigned by the base-class will be 1.0. The component developer should override this value in the class definition as appropriate.

7. Naming Convention Best-Practices

7.1 Purpose

Naming conventions are outlined in the standard to help support best practices and conformity within the standard. These naming conventions are NOT required for development, but will be adopted wherever the standard defines objects, classes, methods, variables, etc.

7.2 Convention

- **Variable capitalization:** camelCase (e.g., yourVariable) or Hungarian (e.g. intYourInteger)
- **Object/Class capitalization:** proper (e.g., MyObject, ThisClass)
- **Private variable:** prefixed with underscore (e.g., _nextCell)
- **Public variable or property:** no prefix (e.g., personId)

8. Base-class Constructor

8.1 Use of Constructor

The base-class constructor will be used to define variables that are scoped to each specific component. These variables will have the following properties and standards:

(a) ***No*** variables will be page scoped. All variables will be scoped to the base class, or the mnemonic namespace.

(b) Public variables are intended to be used by the inheriting component, NOT the page. The page should never access public or private component variables. Instead, the page will only interact with the component via component properties (e.g. componentInstance.getProperty("data")).

(c) Any variables required by the base class implementation that are not defined in this specification, should be defined as private variables (i.e. start with an underscore, "_").

(d) Store persistent data in public variables that the inherited class will be expected to use.

(e) Variables in the base-class will conform to best-practice naming conventions.

(f) Variables are considered "public" if they are defined below and intended to be accessed and used by the inheriting component class. All public component variables will be scoped to the object using 'this'. For example "**this.variableName**".

(g) Variables are considered "private" if they are: ***not*** defined public variables; intended only to support base-class functionality; not intended for use by an inheriting component class. All private variables must be declared in the instance scope ("this.") and must start with "_base". For Example, "this._baseGlobalId=generateUUID()".

8.2 Public Variables

The following public variables will be implemented by the base-class to support persistent data storage and access. This is required to allow methods, events and properties to access persistent data without passing it to each interface call.

8.2.1 MPage.Component.prototype.data

Name: data

Data Type: variable (specific to each component)

Use: The data variable will store data returned from the loadCcl() method in the standard base-class implementation of loadData(). It can then be accessed by render() when drawing the component.

8.2.2 MPage.Component.prototype.options

Name: options

Data Type: object

Read-only: This value is set by the framework for each component instance and can be read, but should not be updated.

Use: The options object contains any configuration parameters stored as a JSON compliant object structure.

Example:

```
//format of data set by framework
this.options = {"days":1,"events":"Labs"};

//use by component developer
this.cclParams[4] = "DAYS=" + this.options.days + ",EVENTS=" + this.options.events;
```

8.2.3 MPage.Component.prototype.cclProgram

Name: cclProgram

Data Type: string

Use: The name of the CCL program from which data will be retrieved if the default loadData() functionality is utilized.

Example:

```
//set ccl program
this.cclProgram = "1_univ_wa_get_labs"
```

8.2.4 MPage.Component.prototype.cclParams

Name: cclParams

Data Type: array

Use: An array containing a list of values that will be combined and send to XMLCclRequest if cclProgram is also populated with a value. If a parameters value is a number, it should be set as a number. If it is a string, it should be a string. When the request parameter string is created by loadCcl() the cclParams string will be joined together and into a comma delimited string. If the parameter is a string, it will be quoted with a carat ^.

Example:

```
//set ccl program
this.cclParams[0] = "MINE";
this.cclParams[1] = this.getProperty("personId");
this.cclParams[2] = this.getProperty("encntrId");
this.cclParams[3] = 100;
this.cclParams[4] = “Labs”

//resulting request string send to XMLCclRequest
var sRequest = “^MINE^, 12345.00, 65667.00, 100, ^Labs^”;
```

8.2.5 MPage.Component.prototype.cclDataType

Name: cclDataType

Data Type: string

Use: The data type that the cclProgram will return. This determines how loadCcl will process and store the data in the this.data public variable.

Accepted Value: JSON, XML, TEXT

Default Value: JSON

Example:

```
//set ccl program
this.cclDataType = “JSON”;
this.cclDataType = “XML”;
this.cclDataType = “TEXT”;
```

8.2.6 MPage.Component.prototype.baseclassSpecVersion

Name: baseclassSpecVersion

Data Type: number (real)

Read-only: YES, this value is set by the framework for each component instance and can be read, but should not be updated.

Use: This variable will store the standard version number that the base-class is compatible with (such as 1.2, 2.1, etc.). A component base-class will be expected to implement the functionality required to support this standard. This value will be set by the framework and should be considered read-only. The component developer can check this version support to identify if there are feature that are not supported that should be inactivated.

Example:

```
//format of data set by framework
MPage.Component.prototype.baseclassSpecVersion = 1.2;

//use by component developer
if (this.baseclassSpecVersion >= 2.0){
	//do something with supported functionality
} else {
	//alert user that functionality not supported
}
```

8.2.7 MPage.Component.prototype.componentMinimumSpecVersion

Name: componentMinimumSpecVersion

Data Type: number (real)

Default Value: 1.0 (the base-class will set a default value of 1.0 which is the minimum).

Read-only: YES, this value is set as part of the component class definition and can be read by the framework, but should not be updated.

Use: This required variable will be set by the component developer when the component extends the base-class. It can be used by the framework during render time, to identify if the component and framework meet minimum compatibility requirements and if the component should be loaded. This value should be set when the component is defined so that each instance will start with the value defined.

Use Requirements:

(a) This value should ONLY be set as part of the component class definition.
(b) This value should NOT be updated or set during the component instance creation.

Example:

```
//defined for the class definition
univwa.MyComponent.prototype.componentMinimumVersion = 1.0;
```

9. Base-class Methods

9.1 MPage.Component.prototype.init ( ) { }

Name: init

Use: Initialize the component internal variables prior to loading data or rendering. The component should NOT attempt to render during the init() subroutine as the DOM element may not exist.

Base-class Requirements:

(a) The init() method will be called prior to loadData() for each component. However, the sequencing of init() between components should not be assumed.

Override Optional: the init() method can be optionally overridden as needed by the component developer.

Override Requirements:

(a) The init() function can assign internal variables.
(b) The init() function CANNOT make any data requests.
(c) The init() function CANNOT attempt to render to the page.

Return Value: none

Parameters: none

Example:

```
//override init() function to define the ccl variables
univwa.MyComponent.prototype.init = function (){
        this.cclProgram = “univ_wa_get_labs”;
        this.cclParams[0] = “MINE”;
        this.cclDataType = “JSON”;
};
```

9.2 MPage.Component.prototype.getTarget ( ) { }

Name: getTarget

Use: Called by the render() function to identify the actual target that should be updated during rendering. This function should only be called in the render() subroutine. It should not be assumed that the DOM object exists prior to the render() function being called.

Base-class Requirements:

(a) If the target element does not exist, then undefined should be returned.
(b) getTarget() should not generate an element on the page as this is a violation of the black-box design principle that limits component DOM interaction to the pre-defined page scope.

Override: NO, this contains standard base class functionality that should not be overridden by the component.

Return Value: DOM object

Parameters: none

Example:

```
//override init() function to define the ccl variables
univwa.MyComponent.prototype.render = function (){
        var oDiv = this.getTarget();
        oDiv.innerHTML = "Hello World";
};
```

9.3 MPage.Component.prototype.loadData (callback(this){}) { }

Name: loadData

Use: The loadData() function is used to asynchronously retrieve data, store it in the component and return control to the page for further execution.

Base-class Requirements:

(a) The loadData() function will be called by the page after init() has been called for that component.
(b) In the base-class implementation, loadData () will contain default functionality to
    a. Call loadCcl() if the cclParams and cclProgram variables are defined
    b. The input parametes will be cclParams, cclProgram and cclDataType. The callback function will assign the data reponse to the this.data public variable.
    c. Call the callback function with a reference to the current object once all asynchronous data calls within that component have completed

Optional Override: The loadData() method can be left to use the default functionality or optionally overridden as needed by the component developer.

Override Requirements:

(d) All data calls must be asynchronous
(e) After the last data-call is completed, loadData must call the callback function.
(f) All CCL based data request, must be made via loadCcl(). This is required so that a page that uses the web services instead of XMLCclRequest can still process the request.

(g) If loadCcl() is called multiple times, the callbacks of each loadCcl() request must be chained together so that the loadData() callback function is executed last.

Return Value: none

Parameters: 1

Name: callback

Use: The callback parameter is a function that should be called after the component has asynchronously loaded its data. This is critical functionality when using Ajax to load data as the main page program's thread will have ended and will not resume control otherwise. The actual callback function would be generated and controlled by the page, not the component. The component only calls that function.

Data Type: function (this) {}

this: the parameter passed returned to the callback must be a reference to the component object instance.

Required: yes

Example:

```
//override loadData() function to call loadCcl()
univwa.MyComponent.prototype.loadData = function ( callback ){
        loadCcl(
                "univ_wa_get_rad",
                ["MINE"],
                function(data){
                        this.data[0] = data;
                        callback();
                },
                "XML"
        );
};
```

9.4 MPage.Component.prototype.loadCcl (cclProgram, cclParams, callback(data){ }, [cclDataType]) { }

Name: loadCcl

Required Use: loadCcl() must be used for all calls to retrieve data from a CCL program.

Use: This function will work very similar to loadData(), but is intended to call load CCL and abstract the process of making Ajax calls to CCL. If a user wishes to call a CCL program to retrieve the data, they can do so without the complexities of managing how XMLCclRequest or web services calls are managed.

Base-class Requirements:

a. loadCcl() will be called by the base-class implementation of loadData().
b. loadCcl() will make an asynchronous XMLCclRequest or a web services Ajax call for the program and parameters passed in.

c. loadCcl() will process the CCL responseText based upon the cclDataType that is passed in. In the case of JSON or XML, the response will be interpreted as an object before being passed back as the callback data parameter. If the cclDataType is TEXT, the responseText value will be returned directly as the data parameter in the callback.
d. loadCcl() will call the callback function (with one input parameters – the data) when the XMLCclRequest call is completed (as the last and final step of the Ajax callback).

Override: NO, this contains standard base-class functionality that should not be overridden.
Return Value: none
Parameters: 4

Name: cclProgram
Use: The CCL program to be called
Data Type: string
Required: yes

Name: cclParams
Use: The CCL parameters to be sent
Data Type: array
Required: yes

Name: callback
Use: The callback parameter is a function that should be called after the XMLCclRequest has returned and completed it functionality.
Data Type: function (data) {}
    data: The CCL data that is returned in the callback
Required: yes

Name: cclDataType
Use: The data type (TEXT, JSON or XML) that the cclProgram will return. This determines how loadCcl will process the data before returning it as the callback data parameter
Data Type: variable
Required: no (default value is JSON)

Example:
```
//override loadData() function to call loadCcl()
univwa.MyComponent.prototype.loadData = function ( callback ){
        loadCcl(
                "univ_wa_get_rad",
                "^MINE^",
                function(data){
                        this.data = data;
                        callback();
                },
                "JSON"
            );
        };
```

9.5 MPage.Component.prototype.render () { }

Name: render

Use: The render() function is called after data has been loaded and stored locally in the component object to render the content to the page.

Base-class Requirements:

(a) The render() function should be called by the page immediately after the loadData() function has returned so that it can render its content as soon as possible

(b) In the base-class implementation, render () will check the type of the this.data variable and based upon its type:

  a. String: render it to the component's main DOM element as defined by this.target
  b. Non-string: render a standard message such as "Component render() function not defined")

(c) Render() should access the DOM element by calling getTarget();

Override Requirements:

(a) Render() should only write to the target element (with the exception of tooltips and popup layers – see section 4.4.2).

(b) Render() should access the DOM element by calling getTarget();

Return Value: none

Parameters: none

Example:

```
//override render to draw to the page
univwa.MyComponent.prototype.render = function (){
        var oDiv = this.getTarget();
        oDiv.innerHTML = "Hello World";
};
```

9.6 MPage.Component.prototype.getProperty(name) { }

Name: getProperty

Use: This function is called to retrieve the property values. In general this will be called by the component to retrieve these values from the page. Properties are values that are exposed and managed by the page and are not specific to a single component's functionality. This would tend to be things like person_id, encntr_id, etc. This is also used to access public component properties by the page.

Base class Requirements: The base class must be integrated with the page to help define and provide access to the property values. By default, the base class will not know what the person_id is. The page must provide that information to the base class.

(a) The base class must expose access to page level values via properties.

(b) This is usually done by overriding the standard base class functionality to supply these values, or by developing a custom base class that is designed to work with a specific MPage to pull the values from pre-defined location. Either is acceptable.

(c) The name parameter will conform to the property names defined within the standard. That way, users can reliably expect to get data for a specific name value

(d) If a particular property is not supported on a page, it should return "undefined", but should never throw a run-time error.

(e) If a property is defined on this component instance, that value should be returned instead of the Page variable

Override: NO, this contains standard base class functionality that should not be overridden by the component.

Return Value: variable (the value of the property will be returned)

Parameters: 1

> **Name:** name
> **Use:** The name of the property
> **Data Type:** string
> **Required:** yes

Example:

```
//get a property
var this.cclParams[2] = this.getProperty("personId");
```

9.7 MPage.Component.prototype.setProperty(name, value) { }

Name: setProperty

Use: This function is called to set instance property values. In general this will be called by the page to set the values on thecomponent.

Base class Requirements: The base class must set this value as a public property of the instance. It should NEVER set the value at the page level as components should not be able to affect each-other.

Override: NO, this contains standard base class functionality that should not be overridden by the component.

Return Value: component object reference (allows for chaining)

Parameters: 2

> **Name:** name
> **Use:** The name of the property
> **Data Type:** string
> **Required:** yes

> **Name:** value
> **Use:** The value of the property
> **Data Type:** variable

Required: yes

Example:

```
//set a property
this.setProperty("headerSubTitle", "(Last 30 days of data)")
        .setProperty("headerTitle","Laboratory");
```

9.8 MPage.Component.prototype.resize(width, maxHeight) {}

Name: resize

Use: This method is designed to be overridden by the component to define what happens when the component is resized by the page or window.

Base-class Requirements:

(a) By default, the base-class function should return null.
(b) The framework should fire resize for each component if the container target elements size changes or may have changed (i.e., page resized).

Override Requirements:

(a) The resize() events should only be used to trigger re-rendering of the component to optimize sizing.
(b) The resize() event should never trigger data requests due to the impact to page performance.

Return Value: none

Parameters: 2

Name: width
Use: the actual width of the component DOM object
Data Type: number
Required: yes

Name: maxHeight
Use: the maximum height the component DOM object will support before applying scrolling. If this parameter is null then there is no maximum height applied to the component DOM object and its contents will take up as much space as necessary.
Data Type: number or null
Required: yes

Example:

```
//define what happens on resize
univwa.MyComponent.prototype.resize = function(width, maxHeight) {
        var oDiv = getElement();
        if (maxHeight > 400){
                //resize the component content
```

```
		}
	}
```

9.9 MPage.Component.prototype.unload() {}

Name: unload

Use: This method is designed to be overridden by the component to define what happens when the component is unloaded as the page closes.

Base-class Requirements:

(a) By default, the function should return null.

(b) The framework should fire unload when the page is unloading or the component is being removed from page scope.

Override Requirements:

(a) The unload () events should be used to clean-up any memory (DOM or JavaScript) as appropriate.

Return Value: none

Parameters: none

Example:

```
	//define what happens on unload
	univwa.MyComponent,prototype.unload = function() {
		myObjectData = nothing;
		var oDiv = getElement();
		var arrItem = oDiv.getElementsByClass("element_class");
		arrItem[0].onClick = nothing;
	}
```

9.10 MPage.Component.prototype.getComponentUid() { }

Name: getComponentUid

Use: This function is called to retrieve a UID string that is unique to each component instance. It will be used by that component instance when generating unique names to be compliant with the naming convention rules. This will usually be concatenated with a description to create unique element ids, names, classes, etc.

Base class Requirements: The base-class must implement this method to generate a sufficiently long and unique identifier for each component instance.

Override: NO, this contains standard base class functionality that should not be overridden by the component.

Return Value: string

Parameters: none

Example:

```
	//get a property
	var sElementId = this.getComponentUid() + "_myelement_1";
```

9.11 MPage.Component.prototype.throwNewError(descr, [error]) { }

Name: throwNewError
Use: This function is used to raise new errors that the framework can handle. Unlike the standard throw new Error() mechanism, these are not real JavaScript run-time errors. Instead, these are logical errors that the framework will capture and act on. This has the advantage that it can be called from asynchronous processes such as setTimeout(), XMLHttpRequest, XMLCclRequest, DOM events, etc.
Base class Requirements: The base-class must implement this method to such that the framework will listen to and handle these events as they arise.
Override: NO, this contains standard base class functionality that should not be overridden by the component.
Return Value: string
Parameters: 2

> **Name:** descr
> **Use:** the description of the error that will be logged by the framework
> **Data Type:** string
> **Required:** yes
>
> **Name:** error
> **Use:** an actual JavaScript error object that was captured in a try/catch routine
> **Data Type:** error object
> **Required:** no

Example:

```
//get a property
var sElementId = this.getComponentUid() + "_myelement_1";
```

9.12 MPage.Component.prototype.refresh() { }

Name: refresh
Use: This function is used by either the framework or the component to refresh the component. This will cause the component to reload based on the current state of the component's options, properties, cclProgram variable and cclParams.
Base class Requirements: The base class implementation MUST:

(a) Clear the existing DOM (including the target element, header sub-titles, etc.)
(b) Call init(), loadData() and render() in the correct sequence

Override: NO, this contains standard base class functionality that should not be overridden by the component.
Return Value: none
Parameters: none

Example:

```
var me = this;
oButton.onclick = function(){
    me.refresh();
}
```

10. Base-class Properties

Properties are values that are exposed by and managed by the page and are not specific to a single component's functionality.

10.1 Property vs. Public Variables

Properties are intended to be an interface between the page and the component. They are accessible from both sides of the "black-box" and define the interface specification. Public variables are intended to be used by the inheriting component, NOT the page. The page should never access public or private component variables.

10.2 Property Access Methods

Component properties will be accessed by the component using getProperty and setProperty as defined in sections 9.8 and 9.9 respectively.

10.3 Property Management

Properties must be exposed to the component using some mechanism. This is the responsibility of the page. The page must expose the property value by defining the getProperty and setProperty method functionality by overloading the component definition after the object is instantiated.

10.4 Standard Properties

The following property names will be used. They will follow the camel case naming convention of public variables and properties.

Name: personId
Data Type: number
Use: access the person Id
Component Access: read

Name: encounterId
Data Type: number
Use: access the encounter Id
Component Access: read

Name: userId
Data Type: number

Use: access the user Id
Component Access: read

Name: headerTitle
Data Type: string
Use: the title displayed in the header of that component if supported by the page
Access: read/write

Name: headerSubTitle
Data Type: string
Use: the sub-title displayed in the header of that component if supported by the page
Access: read/write

Name: headerShowHideState
Data Type: boolean
Use: set or retrieve the show/hide state if supported
Access: read/write

Name: headerOverflowState
Data Type: boolean
Use: set or retrieve if the component div is currently overflow
Access: read/write

Name: compSourceLocation
Data Type: string
Use: The absolute path to the root level of the component source location. This can be used by a component to determine where files are located including images, for supplying absolute referenced paths (i.e., src="' + getProperty("compSourceLocation") + '\my_comp\my_image.png").
Access: read

Name: maxHeight
Data Type: number or null
Use: The maximum height the component DOM element can be before vertical scrolling will be applied on that element
Access: read

11. Event Mechanism

11.1 Event Handling

There are two ways to manage an even structure: either as null functions that can be overridden, or using an event handler mechanism. The former option is simplest to implement and use. The later supports registering multiple events with a single event handler.

Each component must interact directly with the framework for event handling, and cross-component events are not supported (based upon the black-box design principle). As a result, an event handler mechanism is not required. Instead, the simpler mechanism of overriding null functions is sufficient and is implemented under the standard.

11.2 Page Level Events

Page level events are fired by the page. This is done by calling the appropriate function for each component. By default, the functions simply return null. However, if a component wishes to listen to a page level event, it should override the functions listed below. These are defined in more detail in the methods section. There are two supported events methods: resize() and unload()

11.3 Custom Component Events

Custom component events allow a component to raise a custom event with a framework. The action that is taken is up to the framework based upon the custom events it supports. Events are fired using the fireCustomEvent() method.

12. Refresh / Reload Functionality

A component can implement its own internal logic to request new data and redraw itself. There is no exposed functionality in the base-class definition to assist with this.

If the framework needs to refresh a component it must do so by calling init(), loadData() and render() again in the correct order. Each component developer should anticipate that this is possible. As such, a subsequent call of either method should reset, but never append the data. This applies to saving an data requested from loadData() or rendering to the page.

13. Component Element Naming Conventions

The following naming conventions are required to insure interoperability in the DOM.

13.1 Element Id

This must be unique to all components on the page. To support this, each element id should consist of the id of the unique element id (see section 6.2.2) concatenated with a unique string specific to that element.

[unique element id]_[id unique to object instance]

Example:
this.getComponentUid() + "_td_1"

13.2 Element Name

This must be unique to the component instance. To support this, each element name should consist of the unique element id (assigned by the page) as well as a unique string used to specify the actual name.

[unique element id]_[generic element name]

Example:
this.getComponentUid() + "_radiobutton"

13.3 Element Classes and Styles

CSS stylesheets should be favored over inline style definitions, for look and feel based styling. Inline styling can be used for functional and layout based styling that should not be modified (such as table styles for layout control). There are multiple use cases where a class would be assigned and each has their own best-practice or standard. The standards are outlined below

13.3.1 Organization Specific

This would apply to classes that are defined by an organization and shared across multiple components for a common look and feel. These classes must be named using the client's mnemonic as a prefix

[mnemonic]_[description of class]

Example:
univwa_ comment

13.3.2 Component Class Specific

This would apply to classes that are specific to a component class. In this case each instance of a component would have the same class for a common look and feel. These classes must be named using the client's mnemonic and the components name as a prefix.

[mnemonic]_[component name]_[description of class]

Example:
univwa_mycomponent_comment

13.3.3 Component Instance Specific

This would apply to specific component instances. This is not always needed as the element instance can be used as a selector prior to the class for CSS definition and DOM manipulation. In the case that it is needed, the object element target id should be used as the prefix.

[unique element id]_[description of class]

Example:
this.getComponentUid() + "_tall_letter"

13.3.4 External Dependent CSS

There are not requirements as this is dependent upon the external library.

Example:

jquery_calendar_smooth

13.3.5 Common CSS

There are not requirements as this is dependent upon the definition of a common class as outlined below.

Example:

abnormal_result

14. Component CSS Naming Conventions

Custom CSS files created as dependencies to a JS component class must meet the following requirements:

(1) CSS selectors cannot affect an element type (i.e., DIV).

(2) CSS selectors must be limited to matched elements that have an id or class that consist of either a specific component or an organization's mnemonic. This should be consistent with the DOM naming convention outlined above.

OK

```
.univwa_comment
.univwa_mycomponent_tall_letter
#component1 .univwa_comment
.univwa_mycomponent div
```

NOT OK

```
table
.mycomment
. mycomponent_tall_letter
#genericid .univwa_comment
.univwa_mycomponent, div
```

15. CCL Naming Conventions

To avoid naming conflicts with the CCL dependencies of a component, CCL script should be named according to a convention:

(1) The CCL script name must be unique to the developing organization, starting with the client mnemonic or a number followed by the mnemonic.

(2) The CCL script should ideally be named to match or be prefixed by the client mnemonic and class name, consistent with the namespace scoping rules (mnemonic.ClassName).

OK

```
univ_wa_my_component_get_data
univ_wa_get_ce_data
1_vcu_get_data
```

NOT OK

```
my_component_get_data
my_component_univ_wa
```

16. JavaScript Dependencies

Components are likely to be dependent upon external JavaScript files. These can range from simple date libraries to complex implementations like jQuery and extJS. In either case, it is expected that the component developer will clearly document the dependencies.

There are no limitations on which JavaScript libraries can be used as dependencies; however, documenting the requirements as well as possible interoperability conflicts is the responsibility of the developer.

The MPage framework must be capable of loading these dependencies when the page is created. They can be hard-coded or dynamically generated.

It is highly recommended that any external files utilized by the component be placed in a namespaced folder to prevent naming collisions between files.

17. CSS Dependencies

Components are likely to depend on both internal and external CSS files. These can include CSS files used by JavaScript libraries. Additionally, these can be CSS dependencies developed specifically for that component.

There are no limitations on which CSS libraries can be used as dependencies; however, documenting the requirements as well as possible interoperability conflicts is the responsibility of the developer.

It is highly recommended that any external files utilized by the component be placed in a namespaced folder to prevent naming collisions between files.

18. Error Handling

A component should attempt to gracefully recover from any runtime error. However, if the error arises and cannot be overcome, it must be bubbled up to the framework. This includes situations where the component has identified an error in a try/catch function, but is not completely resolved (i.e., data is missing, etc.). There are three ways this can be done.

1) The component can allow the **existing error to bubble up**. This is not best practice, but works. However, this can ONLY be done when the current component thread is running under the control of the framework. This should NEVER be done in the following circumstances:
    a. Within an asynchronous AJAX process (i.e., XMLCclRequest.onreadystatechange{}, etc.)
    b. Within an asynchronous setTimeout() or setInternal() call.
    c. Within an asynchronous DOM event (i.e., onClick="")

2) The component can raise a new JavaScript runtime error using "**throw new Error()**". However, this can ONLY be done when the current component thread is running under the control of the framework. This should NEVER be done in the following circumstances:
    a. Within an asynchronous AJAX process (i.e., XMLCclRequest.onreadystatechange{}, etc.)
    b. Within an asynchronous setTimeout() or setInternal() call.
    c. Within an asynchronous DOM event (i.e., onClick="")

3) **BEST PRACTICE**. The component can call the method **this.throwNewError()** and pass in a description. This can be done at any-time and has the following advantages:
    a. It can be called during any process, including asynchronous processes that are now allowed above.
    b. It can include non-runtime errors.
    c. It will allow processing to continue if appropriate within the components current method that is being executed.

The component should NEVER directly alert the user that an error has occurred using the alert() function. This interrupts processing logic and affects the page as a whole. Instead, any visual alert should be handled by the page or embedded in the components display as appropriate.

Any framework, must be able to handle errors that bubble up from the component, and should do so gracefully (i.e., alerting the users, logging the error, etc.).

<u>The framework must be able to handle errors that bubble up like this (assuming they occur in synchronous processing when the init(), loadData() or render() methods are called).</u>

```
try {
	for (var i = 1; i < oData.length; i++){
		oData[1].hello = oData[0].world.value;
	}
} catch (err) {
	If (oData != undefined){
		throw new Error("Error processing data: " + i + " " + oData.length);
	} else {
		throw new Error("Data response is undefined.");
```

```
            }
    }
```

19. Common CSS

In order to support a common look and feel amongst components, the standard will define and support common CSS classes that can be used by component developers. To be compliant with the standard, each framework developer must support these classes, making them available to the component developer. The standard will define the class name and the appropriate usage of each class. The first release of common CSS classes will occur with the 1.1 standard.

19.1 Result Normalcy Classes

The following classes are used to properly display results including color coding and out of range indicators (arrows, exclamation marks, etc.). These classes should be defined for use with result values only and generally are used to define a common look and feel consistent with PowerChart. There are two use cases.

The Result Normalcy Classes can be used with a single element (span, div, etc.) to define the coloring of the included values:

```
<span class='res-high'>100</span>
```

The Result Normalcy Classes can be combined with the Result Sub-Classes to define the coloring of the value as well as an optional indicator icon element which will include a background image for rendering:

```
<span class='res-high'>
  <span class='res-ind'>&nbsp;</span>
  <span class='res-val'>100</span>
</span>
```

Class Name	Applies To	Usage	CSS Framework Example	Usage Example	Display Example
Result Normalcy Classes					
res-high	Any Element	Should be applied to results with a normalcy value of HIGH	.res-high { color: orange; }	<span class="res-high">100</span>	100
res-low	Any Element	Should be applied to results with a normalcy value of LOW	.res-low { color: blue; }	<span class="res-low">50</span>	50
res-severe	Any Element	Should be applied to results with a normalcy value of	.res-severe { color: red;	<span class="res-severe">150</span>	**150**

		EXTREMEHIGH, PANICHIGH, EXTREMELOW, PANICLOW, CRITICAL, POSITIVE and VABNORMAL	font-weight:bold; }		
res-abnormal	Any Element	Should be applied to results with a normalcy value of ABNORMAL	.res-abnormal { color: orange; }	<span class=”res-abnormal”>Weak</span>	Weak
res-normal	Any Element	Should be applied to results other than those listed above	.res-normal { color: black; }	<span class=”res-normal”>Average</span>	Average
Result Sub-Classes					
res-val	sub-element of one of the normalcy indicators	Should be applied to the result value of a sub-element of one of the above items. Used in conjunction with res-ind to apply coloring. Must be defined for each normalcy class above.	.res-low .res-val { color: blue; }	<span class=”res-normal”> <span class=”res-val”>50</span> <span class=”res-ind”></span> </span>	50↓
res-ind	sub-element of one of the normalcy indicators	Should be applied to the result icon of a sub-element of one of the above items. Used in conjunction with res-val to apply icons next to result values (arrows, etc.). Must be defined for each normalcy class above.	.res-severe .res-ind { background-url(“…”); }	<span class=”res-severe”> <span class=”res-val”>150</span><span class=”res-ind”></span> </span>	**150 !**

19.2 Header and Row Classes

The following classes are used to properly display headers and rows. These are included to help maintain a consistent coloring scheme amongst components including section and table headers and alternating rows. These can be used to support both tables and other elements that consist of headers and alternating rows.

Implementation Hint:

Frameworks that wish to style the color of rows and headers differently based upon the “theme” of the component or the page can:

(a) Flex the CSS definition for a page level theme, or

(b) Define a component level class that will be applied to the component frame (by the framework) that will flex the coloring of each individual component.

Class Name	Applies To	Usage	CSS Framework Example	Usage Example

hdr	Any Element	Should be applied to table headers or section headers where you wish to have them styled with a background color consistent with the overall page. This is usually a lighter version of the main component header color (which is controlled by the framework)
		
		DOES NOT REQUIRE THE USE OF TH ELEMENTS FOR CELLS (however, this is best practice).
			.hdr {
			  background: light green;
			}
			
			.hdr > td .hdr > th {
			  background: light green;
			}
			
			.blue .hdr {
			  background: light blue;
			}
			
			.green .hdr {
			  background: light green;
			}
				<div class="hdr">…</div>
				
				<tr class="hdr">…</tr>

even	Any Element	Should be applied to even row elements (<u>not</u> counting any header rows), If you wish to color alternating rows.
		
		CAN BE USED WITH TALBES OR NON-TABLE ELEMENTS.
			tr.even {
			  background: gray;
			}
				<div class="even">…</div>
				
				<tr class="even">…</div>

odd	Any Element	Should be applied to odd row elements (<u>not</u> counting any header rows), IF you wish to color alternating rows.
		
		CAN BE USED WITH TALBES OR NON-TABLE ELEMENTS.
			tr.odd {
			  gray: white;
			}
				<div class="odd">…</div>
				
				<tr class="odd">…</div>

19.3 Label and Value Classes

The following classes are used to properly display labels and values (excluding result values, which are defined above). These are included to help maintain a consistent coloring scheme among components that have label-value pairs. This definition is fairly flexible and can be applied to many possible elements within a component.

Class Name	Applies To	Usage	CSS Framework Example	Usage Example	Usage Example

label	Any Element	Should be applied to an element that is being used as a label (i.e., a drug name, where the value is the	.label {
			  color: gray;
			  font-size: 10pt;
			}
				<span class="label">
				Tylox
				</span>
					Tylox

		detailed display line, or an event set name).			

value	Any Element	Should be applied to an element that is being used as a value (i.e., a drug detailed display line, where the drug name is the label).
		
		SHOULD NOT BE USED WITH RESULT VALUES (see the Result Normalcy Classs)
			.value {
			color: black;
			font-size: 11pt;
			}
				<span class=”value”> 100 mg Oral Two Times Daily for pain </span>	100 mg Oral Two Times Daily for pain

Appendix 1 – Functional Map

```
MPAGE FRAMEWORK NAMESPACE
MPage = {};

COMPONENT BASE-CLASS CONSTRUCTOR
MPage.Component = function() {};

COMPONENT BASE-CLASS PUBLIC VARIABLE
MPage.Component.prototype.options = null;
MPage.Component.prototype.data = null;
MPage.Component.prototype.cclProgram = null;
MPage.Component.prototype.cclParams = null;
MPage.Component.prototype.cclDataType = “JSON”;
MPage.Component.prototype.componentMinimumSpecVersion = 1.0;
MPage.Component.prototype.baseclassSpecVersion = null;

COMPONENT BASE-CLASS PUBLIC METHODS (OVERRIDABLE METHODS)
MPage.Component.prototype.init = function() {};
MPage.Component.prototype.loadData = function( callback( this ){} ) {};
MPage.Component.prototype.render = function() {};

COMPONENT BASE-CLASS PUBLIC METHODS (OVERRIDABLE EVENTS)
MPage.Component.prototype.unload = function(){ return null; };
MPage.Component.prototype.resize = function( width, height ){ return null; };

COMPONENT BASE-CLASS PUBLIC METHODS (NOT OVERRIDABLE)
MPage.Component.prototype.loadCCL = function( cclProgram, cclParams, callback(data){},
       [cclDataType] ) {};
MPage.Component.prototype.throwNewError = function( descr, [error] ) {};
MPage.Component.prototype.getProperty = function( name ) {};
MPage.Component.prototype.setProperty  = function( name, value ) {};
MPage.Component.prototype.getTarget = function() {};
MPage.Component.prototype.refresh = function(width, maxHeight) {};
```

<u>CLIENT MNEMONIC NAMESPACE</u>

```
Mnemonic = {};
```

<u>CUSTOM COMPONENT CONSTRUCTOR AND INHERITANCE</u>

```
Mnemonic.MyComponent = function () {};
Mnemonic.MyComponent.prototype = new MPage.Component();
Mnemonic.MyComponent.prototype.constructor = MPage.Component;
Mnemonic.MyComponent.prototype.base = MPage.Component.prototype;
```

<u>REQUIRED PROPERTIES</u>

```
personId
encounterId
userId
compSourceLocation
```

<u>OPTIONAL PROPERTIES</u>

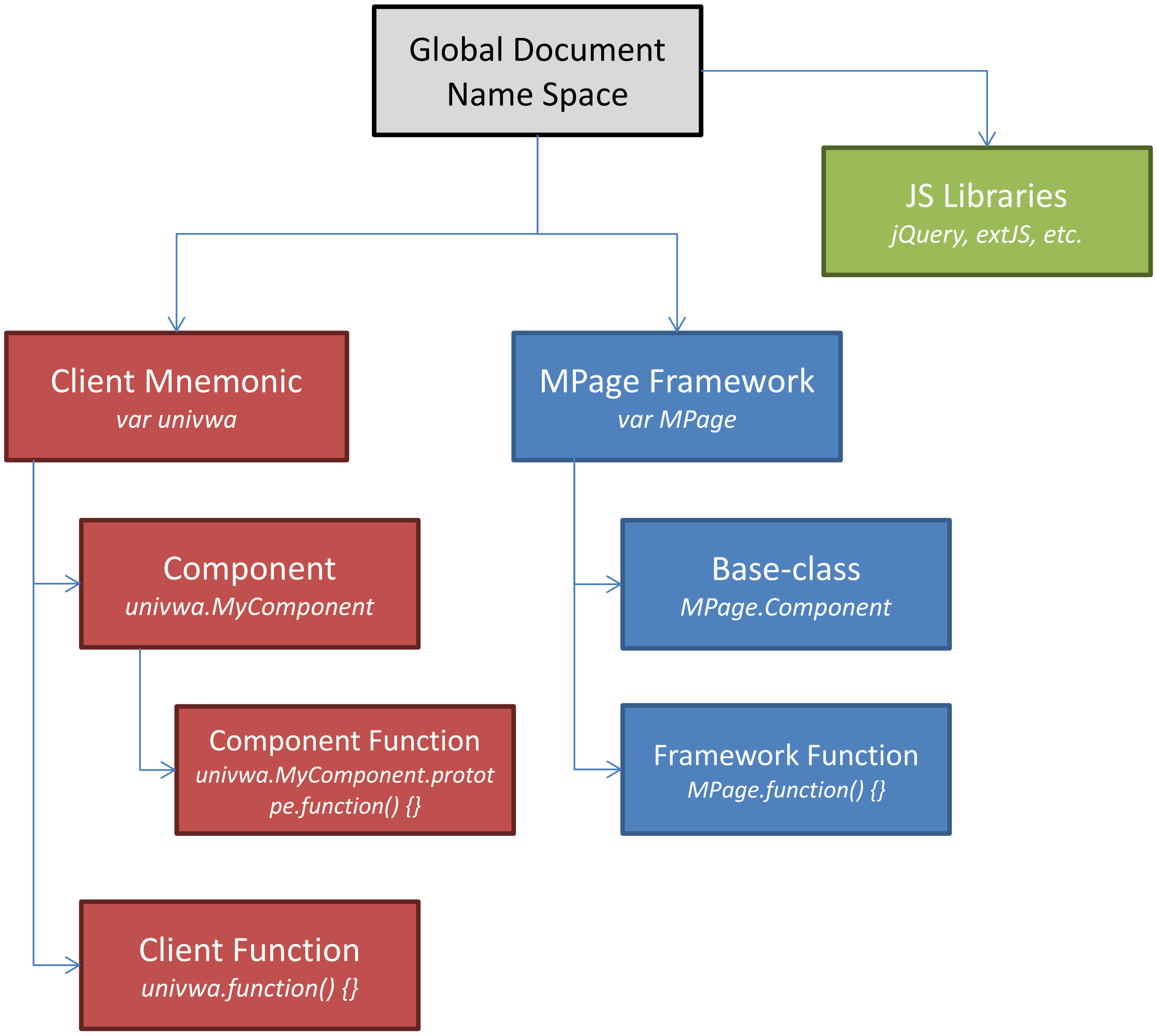
```
PPRCode
positionCd
location (current users device)
defaultLocation (WTS)
appName
headerTitle
headerSubTitle
headerShowHideState
headerOverflowState
headerToolbar
maxHeight
```

<u>COMMON CSS CLASSES</u>

```
res-low
res-high
res-abnormal
res-normal
res-severe
res-ind
res-value
hdr
even
odd
label
value
```

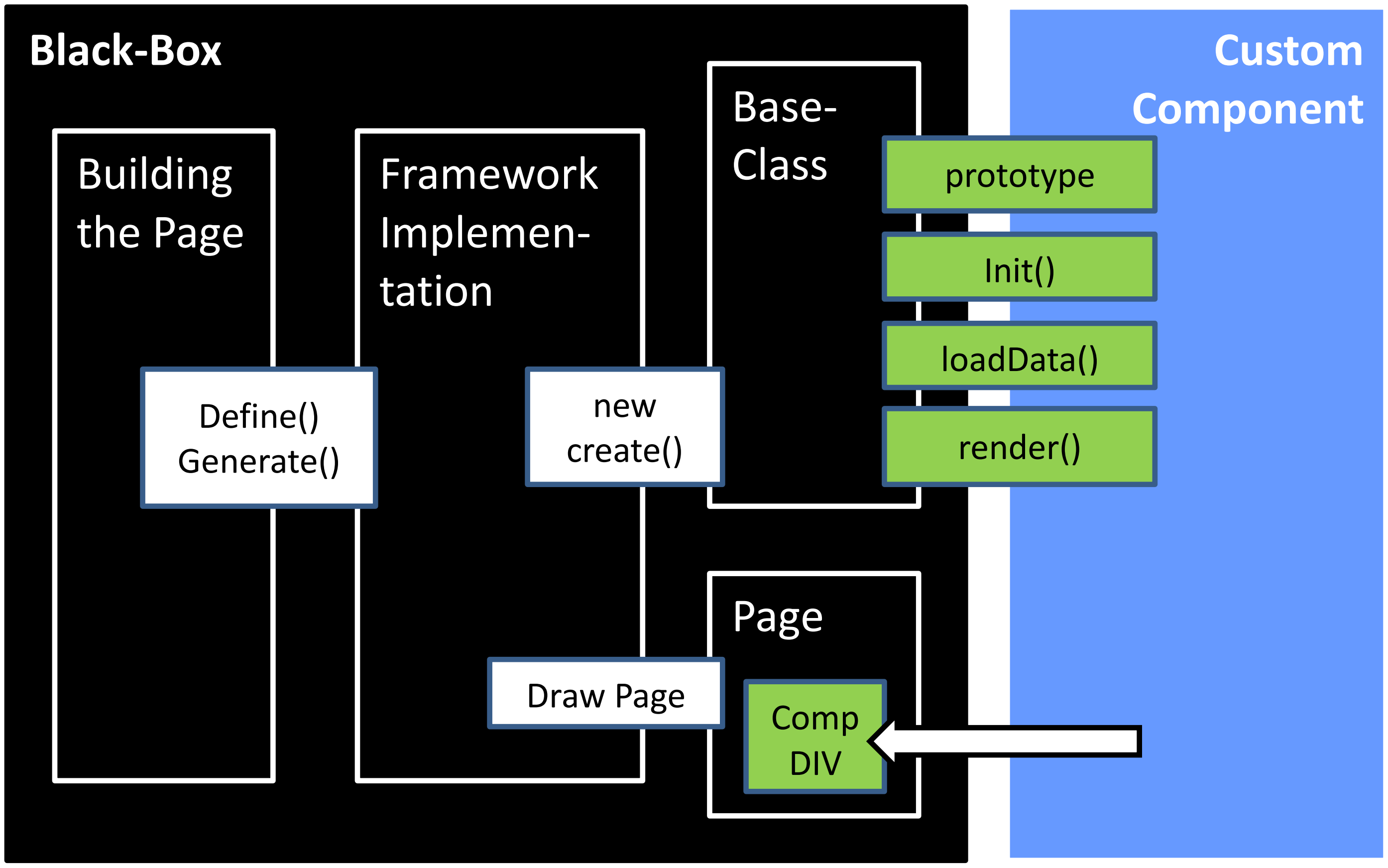
Appendix 2 – Name Space Map

The following diagram outlines the name-spacing that must be followed when developing components a framework or defining client specific functions, etc. A full description is found in section 5.



Appendix 3 – Framework Black-Box Design Principle

The following diagram illustrates the black-box design principle that should be followed when developing a framework. The interfaces that are exposed from the framework are limited to those items in green that consist of the well-defined interfaces described in this standard. How each specific framework is implemented is NOT defined by the standard and is up to the framework developer. This includes (in white): How the framework generates or creates the component instances, how the framework draws the page, and how the framework allows developers, analysts or users add a component to a page.



Appendix 4 – CSS Classes

DATA CLASSES (focus on safety)
mpage_result_abnormal (font / color / image)
mpage_result_high (font / color / image)
mpage_result_low (font / color / image)
mpage_result_critical (font / color / image)
mpage_result_normal (font / color / image)
mpage_allergy_severe (font / color / image)
mpage_allergy (font / color / image)

LAYOUT CLASSES (general look and feel)
mpage_table (background color)
mpage_table_row_alternate (background color)
mpage_highlight (background color)
mpage_tooltip (frame)
mpage_tooltip_body (background color)
mpage_tooltip_header (background color)

Appendix 5 – Best Practices

The following represent recommended best practices in both the implementation of components as well as an MPage framework.

Component Implementation Best Practices

1. In most cases, components should consist of one JS file, one CSS file and one CCL script.
2. Whenever possible, components should rely on the base-class loadData() function. This would generally be the case, when a single CCL call is required to retrieve data. In this case, the CCLProgram and CclParams variables should be set in the init() method.
3. CCL programs that act as a component's data service, should in general have 5 inputs: OUTDEV, personId, encntrId, userId, options. This is the standard supported by the base-class if the CclParams function is not modified.
4. Components should override the init() method for any pre-rendering as well as to set the CCLProgram and CclParams values for loadData().
5. If components share data, they should do so using a collection object to synchronize the data. The collection object would be instantiated as a singleton object. Each component that worked with that collection would have the init() function overloaded to register with the component and the loadData() function designed to synchronize data with the collection.
6. In general, event synchronization between components should be avoided as it creates complex dependencies.
7. Common functions shared across an organization's components should be scoped to the client's mnemonic namespace (i.e., univwa.doThis()).
8. If JavaScript libraries are required that use the $ namespace, the component should attempt to implement them in safe mode to avoid conflict. This usually means using an alternate namespace such as "document.id" in MooTools or "jQuery" in jQuery
9. Component developers should attempt to avoid using JavaScript libraries that broadly pollute the global namespace by creating functions at the global level. Additionally, libraries that overload standard JavaScript functions should also be avoided.
10. Avoid inline images that have a standard class.

MPage Framework Implementation Best Practices

1. The MPage framework should render the DOM container for an object before the object is instantiated and create() or init() is called.
2. The MPage framework must make properties available via GetProperty and SetProperty. As such, there should never be a need within a component to read a variable at the MPage object or page global scope (i.e., MPage.personId or personId). This would violate the black-box design principle.