SAND2014-20572R

US NDC Modernization

USNDC2014-xxxx
Unclassified Unlimited Release
December 2014

US NDC Modernization Iteration E2 Prototyping Report: User Interface Framework

Version 1.1

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

USNDC2014-xxxx
December 2014

US NDC Modernization Iteration E2 Prototyping Report: User Interface Framework

Jennifer E. Lewis
Melanie A. Palmer
James W. Vickers
Ellen M. Voegtli

Version 1.11
Sandia National Laboratories
P.O. Box 5800
Albuquerque, New Mexico 87185

ABSTRACT

During the second iteration of the US NDC Modernization Elaboration phase (E2), the SNL US NDC Modernization project team completed follow-on Rich Client Platform (RCP) exploratory prototyping related to the User Interface Framework (UIF). The team also developed a survey of browser-based User Interface solutions and completed exploratory prototyping for selected solutions. This report presents the results of the browser-based UI survey, summarizes the E2 browser-based UI and RCP prototyping work, and outlines a path forward for the third iteration of the Elaboration phase (E3).

REVISIONS

Version	Date	Author/Team	Revision Description	Authorized by
1.0	9/19/2014	US NDC Modernization Team	Initial Release	M. Harris
1.1	12/19/2014	IDC Reengineering Team	IDC Release	M. Harris

TABLE OF CONTENTS

This page intentionally left blank.

1. OVERVIEW

The US NDC Modernization project statement of work identifies the definition of a modernized US NDC system architecture as a key project deliverable. As part of the architecture definition activity, the Sandia National Laboratories (SNL) project team has established an ongoing, software prototyping effort to support architecture trades and analyses, as well as selection of core software technologies.

During the second iteration of the Elaboration phase (E2), spanning Q3 – Q4 FY2014, the prototyping team developed additional COTS surveys and exploratory prototypes, building on E1 prototyping work related to the User Interface Framework (UIF). This report summarizes the iteration E2 prototyping work and discusses the path forward in E3.

1.1. UIF Definition

The User Interface Framework (UIF) is a software mechanism providing a standard architecture and a set of application programming interfaces (APIs). The goals are to provide a single unified UIF to be used for all internal system displays; support a consistent, modern user interface standard for all system graphical user interface display elements; support a responsive and customizable user interface; and provide a modular, component-based architecture for development, integration, and deployment of extensible GUI components. The constraints within which the UIF must operate include supporting user interface performance requirements, enabling developers to build new UI features via standard APIs, interfacing with other frameworks in the system (e.g. the Object Storage and Distribution mechanism), use of Open Source Software (OSS) and Commercial Off-The-Shelf (COTS) software, and preference for solutions based on open standards.

2. E1 BACKGROUND

During E1, UIF tasks focused on surveying available OSS and COTS software solutions for desktop based user interfaces to determine which software merited further investigation through exploratory prototyping. The survey resulted in selection of the NetBeans and Eclipse Rich Client Platforms (RCPs) as the primary candidates for a desktop based UIF. The NetBeans prototype is documented in the E1 UIF prototyping report [1].

This prototyping effort found that NetBeans supports running on a variety of operating systems; supports integration of independently developed plugins; and provides reusable services such as management of the user interface layout, user settings, storage, plugin framework, and progress reporting. Some of the

features available to the sample NetBeans RCP application included the ability to customize window size, window position, and whether a window was docked or disconnected from the main application. In addition, the user could modify the relative position of windows by dragging them to different snap locations on the floating set of connected display widgets or the user could simply "tear off" any given display widget and move it to a location of their choice anywhere on the screen. The exploratory prototype showed many of the UIF goals are met by NetBeans RCP, including Analyst workspace customization requirements, OSS, and extensibility, making it a potential candidate for the UIF.

3. E2 PROTOTYPING

E2 prototyping work involved continued exploration of Rich Client Platforms (RCP) with exploratory prototyping of the Eclipse RCP and investigating the feasibility of implementing the Analyst interfaces as a browser based User Interface (UI). The primary goal was to select between RCP and browser based UIs for implementing Analyst interfaces, and a secondary goal was to decide between NetBeans and Eclipse as the most viable RCP solution.

3.1. Browser based UI

The UIF prototyping effort devoted some of E2 to learning more about the development of web applications, including rich internet applications (web pages with a "desktop-like" feel). The goal of this effort was to assess the feasibility of incorporating browser based interfaces, as opposed to native desktop applications, into the system architecture prototype. Specifically, the UIF prototyping team sought to gain insight into the following aspects of browser based UIs:

- Performance of a browser based UI compared to a similar desktop based UI
- Development level of difficulty
- User experience of a browser based UI versus a desktop based UI
- Existing standards for developing browser based UIs and frameworks that conform to those standards
- Toolsets that could be leveraged within a browser based UI

The browser based UI prototype aimed to create web pages within a web application framework. These web pages include a waveform display and a signal detection table display. Modifications to the signal detection table should update the signal detection information in the waveform display, and vice versa. This requirement of communication between displays guided the search for

frameworks towards software that provides communication between the elements managed by the framework.

3.1.1. Plotting packages – waveform display

A brief survey was conducted to assess available OSS and COTS charting packages. Waveform packages selected for prototyping needed to support:

- Displaying waveform data with performance meeting system requirements
- Providing the ability to zoom, pan, scroll
- Displaying annotations on waveform data (e.g. signal detection markers)
- Providing an interface for manipulating waveform annotations (e.g. adding/deleting/moving signal detections)
- Providing waveform and waveform annotation metadata via tooltips (or similar mechanism)

The two plotting packages that best meet these requirements are D3 and Highstock. Appendix A contains a list of the other packages included in the survey as well as the rationale for why D3 and Highstock were selected over other packages.

3.1.1.1. Prototyping

Because Highstock and D3 were the most promising of the surveyed plotting packages, these are the two the UIF team incorporated into the exploratory prototype for the browser based UI.

3.1.1.1.1. Highcharts: Highstock

Highstock [2] is a Highcharts product that uses JavaScript to create timeline plots. Highstock can load large amounts of data quickly while still presenting a responsive display, including multiple waveforms, to the user. It also provides annotations that can be used to represent signal detections on waveforms.

A concern that arose with Highstock is that it hides some of the underlying functionality in an attempt to be easier to use. This lack of access to the base features Highstock builds upon can limit a developer's ability to tailor a plot display to specific needs different from the defaults offered with Highstock.

3.1.1.1.2. Data-Driven Documents (D3)

D3 is a JavaScript library that allows binding arbitrary data to a Document Object Model (DOM), and then applying data-driven transformations to the document. D3 minimizes code by abstracting looped calls into one single function call applied to an entire set of data at once. It uses HTML, SVG (Scalable Vector Graphics), and CSS.

Extensive customization is possible using the provided API, and it is relatively easy to start from scratch, read in a set of data, and display it in a graph, plot, shape, etc., using just a few lines of D3 calls in JavaScript. D3 avoids excessive DOM updating, which results in a display that is responsive to user interaction.

3.1.2. Frameworks

The E1 prototyping work with NetBeans demonstrated the benefits of using an established framework for the management of independent user interface components. For the browser based UI work of E2, the UIF prototyping team researched which frameworks for web application development exist and which frameworks best support the system user interface goals.

Many frameworks for developing web-based user interfaces are available. These are the initial framework categories included in the survey:

- Widget framework: These frameworks organize web applications as widgets in a single operating environment (e.g. multiple web pages displayed inside a single web page).
- Plugin: Plugins extend existing browser functionality.
- Toolkit: Toolkits provide the tools developers need to create web-based applications, but lack the framework needed for integrating different components.
- Rich Internet Application (RIA) frameworks: These frameworks are for developing Rich Internet Applications (web applications with many of the characteristics of desktop application software).

Web Application Framework: These frameworks support development of dynamic websites, web applications, web services, and web resources while alleviating overhead associated with common web development activities. Software from each of the categories above was included in the initial survey list. Because this resulted in a large number of available software solutions, the survey focus shifted primarily to software solutions that provided support for:

- Customization of window layout

- Customization of user preferences
- Communication between UI components
- Plugins

The survey also took into account industry presence and was limited to software solutions that have an active developer and user community.

Ozone Widget Framework (OWF) was chosen for prototyping because it provides a customizable desktop-like view of applications within a browser. Each widget is a web page which provides developers the flexibility to develop using the appropriate toolkit for the web page task. In addition, OWF provides a communication channel that these web pages can use to communicate or share data.

SproutCore offers a framework for creating all of the web pages needed for an application. Where OWF provides a container for managing independently developed web pages, SproutCore is tightly integrated into the web pages themselves. SproutCore provides its own web page development language (similar to JavaScript) and enforces a structure that all SproutCore applications must adhere to, including mechanisms for communicating across web pages and writing/reading data to/from a common data store. SproutCore was chosen for prototyping because it allows for exploration of a more fully web-based development framework (as opposed to OWF which manages applications developed independently of OWF itself).

Appendix B contains a list of the packages included in the survey as well as the rationale for why OWF and SproutCore were selected over other packages.

3.1.2.1. Prototyping

The goal of both the Ozone Widget Framework and SproutCore prototyping was to learn more about these frameworks specifically and development of desktop-like web applications in general. The resulting applications for both OWF and SproutCore aimed to demonstrate the customizability of the widgets managed by the framework, including widgets / web pages plotting waveform data and viewing a table of signal detections associated with those waveforms.

The prototyping activities need to support the following features:

- Display of many waveforms simultaneously with little to no noticeable lag in performance when zooming, panning, or otherwise manipulating the data display
- Annotations of waveforms for indicating signal detections

- Demonstrate data and display synchronization using communication between the signal detection table and the waveform display, including dynamic updates to the annotations on the waveforms based on changes to the signal detection table

- Sorting capability

3.1.2.1.1. Ozone Widget Framework

The Ozone Widget Framework (OWF) is "a customizable open-source web application that assembles the tools you need to accomplish any task and enables those tools to communicate with each other" [3]. OWF manages UI components through widgets. Each widget is a web page, and permissions for those widgets can be set on a user or group level.

The widgets can be placed in whatever configuration the user wishes, similar to positioning application windows on a desktop, by adding from a list of installed widgets.

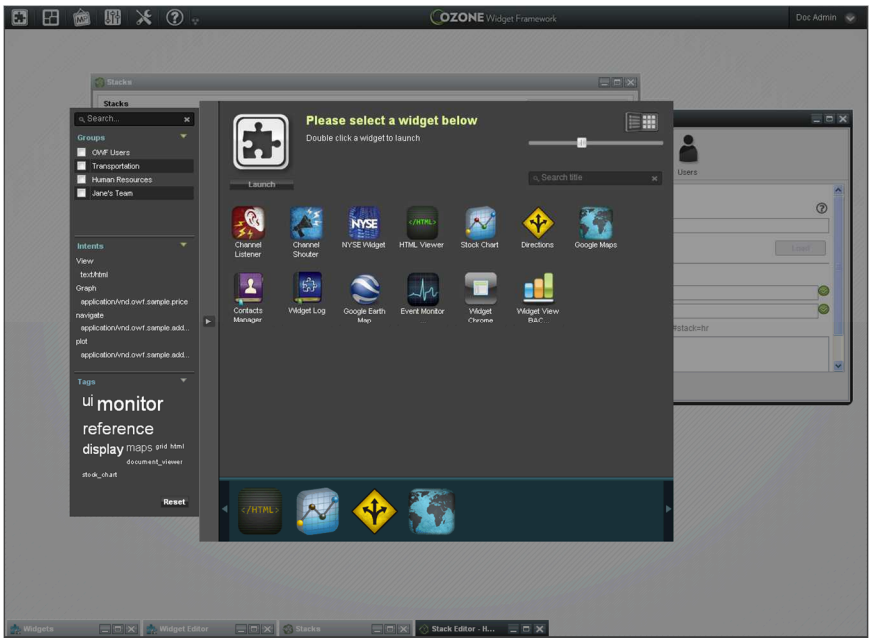

Figure 1 – OWF: Launch menu, selecting a widget

The OWF API is fairly limited and includes communication between widgets, designing the type of window used to display a widget, enabling drag and drop between widgets, creating a log, etc. All of these commands interact with the OWF framework through JavaScript methods. Because JavaScript is ubiquitous within web application development, OWF JavaScript calls can be embedded into widgets built in a variety of languages (e.g. HTML, JavaScript, Java Applets, .Net, Silverlight, Google Web Toolkit, etc.)

Because OWF provides only the framework for managing independent web pages as widgets, the pages within the widgets can be used and tested in a standalone context (without OWF) as well.

3.1.2.1.1.1. Dashboard

Customized dashboards (desktop views) can be created that contain a default set of widgets and define the initial layout of those widgets. Permissions for these dashboards can be set on a user or group level. Dashboard layouts can be customized to have different zones that define window behavior for that zone (e.g. tabbed window versus accordion view for a given zone) and can be created using a drag and drop interface through OWF's dashboard creator.

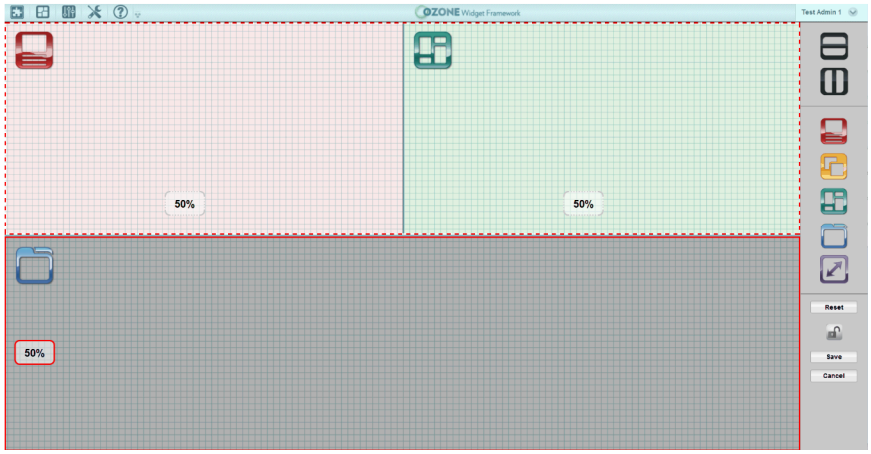

Figure 2 – OWF: Creating a custom dashboard

3.1.2.1.1.2. Communication between widgets

OWF provides a mechanism for widgets to communicate. The provided interface resembles socket communication, but OWF manages the intricacies of opening, closing, and managing the sockets and provides developers with abstract methods of sending and receiving data.

```
// Global function that sends data from the waveform widget to
// the table widget
shout = owfdojo.hitch (this, function() {
      var channel = 'FromWaveform';
      OWF.Eventing.publish(channel, Waveform);
});

// Connect to receive using an OWF Socket Channel
OWF.Eventing.subscribe('FromWaveform',
                        yowfdojo.hitch(this, this.addToGrid));
```

Figure 3 – OWF: Communication example

3.1.2.1.1.3. Administration

Administrators can manage users, user groups, and permissions for widgets and dashboards. In addition, administrators can design and create custom dashboards for different users or user groups.

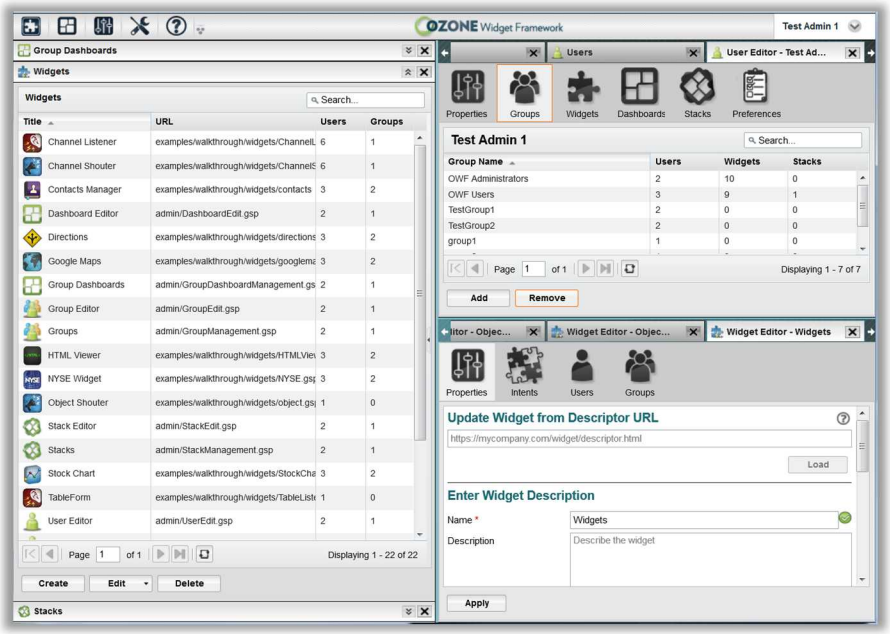

Figure 4 – OWF: Administration panel

3.1.2.1.1.4. Prototype

OWF prototyping used the Highstock plotting package and created a widget to display waveform data and a widget to display a table of signal detections. These widgets communicated with each other such that updates to the signal detection table were received by the waveform display and vice versa. Due to time constraints, only one waveform and one signal detection table entry is displayed in the final version.

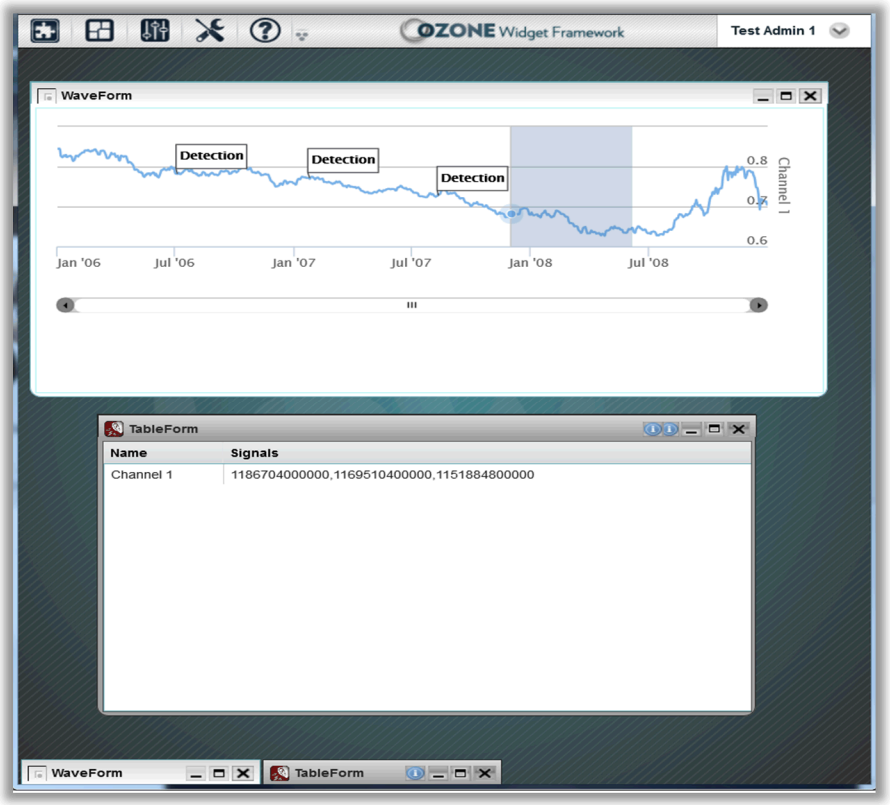

Figure 5 – OWF: Prototyping widgets

3.1.2.1.1.5. Evaluation

OWF provides a flexible framework for managing widgets. Widgets can be independent web pages or they can interact through OWF's underlying communication mechanism. The ability to customize dashboards by user or user group provides a lot of flexibility with how each user manages their individual displays. Should the system require a customizable web-based desktop interface, OWF would be a solid choice.

3.1.2.1.2. SproutCore

SproutCore is "an open-source framework for building blazingly fast, innovative user experiences on the web" [4]. Many web page toolkits that provide dynamic user interactions adhere to the single page paradigm where all data is presented in one, vertically scrollable page. That page could have widgets that have behavior independent from the rest of the page, and that are developed separately, but the result is still a single page. SproutCore offers a "workspace view" that can contain multiple pages that the user can arrange as they wish. For this reason, SproutCore was considered a candidate for further prototyping.

SproutCore introduces its own language and syntax to support object-oriented programming that extends JavaScript objects. Developers create a folder hierarchy conforming to a structure expected by SproutCore. This hierarchy contains files responsible for different functions necessary to populate the web page. At runtime, these files are converted into HTML and rendered to a web page.

SproutCore pushes the entire MVC design pattern into the browser, with the only separate component being the data that is dynamically loaded into the model during application runtime.

3.1.2.1.2.1. Desktop Customization

Desktop customization is not default functionality within SproutCore. Typical window actions for views within an application (e.g. minimize, maximize, and close) are missing; this made desktop customization difficult to accomplish.

3.1.2.1.2.2. Data Binding / Communication

The mechanism that SproutCore utilizes for communication within an application is the Key-Value Observing model, or KVO. A binding connects the properties of two objects so that whenever the value of one property changes, the other property will be changed also. Binding one attribute to another involves a simple binding call.

The easiest type of binding is one-way, which is useful for things like labels. One-way binding should be used when an object needs to be updated based on changes to the object it is bound to, but not vice versa.

```
label: SC.LabelView.extend({
      valueBinding:
SC.Binding.oneWay('NdcTest.waveController.labelValue')
}),
```

Figure 6 – SproutCore: One-way data binding example

This binding call updates a label in the signal detection display when a label value changes in the waveform display. It accomplishes this by referencing a method in `waveController` called `labelValue()`. When a change is made in the `waveController` which requires a change to the label value, a notification is generated that updates the corresponding label in the signal detection display. However, since this is a one-way binding, if the label changes in the signal detection display, it will not change in the waveform display.

SproutCore also supports two-way bindings, such that whenever one side of the binding changes, the other is immediately updated.

During prototyping, a variation of KVO two-way binding was established between a text field in the signal detection table and the name of a waveform in the waveform display. In the table, the editable text field supported a one-way binding to display the station name for a waveform, and clicking a button next to the text field fired an action that would update the name of the station on the waveform display with the new value entered by the user in the text field.

The SproutCore prototype incorporated the D3 plotting package. SproutCore's KVO could not be used to update the signal detection table from the D3 waveform display because the D3 code (wrapped in a function inside a SproutCore view) is not SproutCore-native, it is standard JavaScript. Therefore setting the bindings up the way they are above (a set of calls unique to SproutCore) within the D3 logic would not work, and the bindings had to be limited to SproutCore elements.

3.1.2.1.2.3. Data Management

A unique feature of SproutCore is its DataStore (essentially a local database) that allows for managing object descriptions and relationships easily. This acts as a local-access, in-memory database that runs each time the application is launched. It is customizable and includes such functions as access to a server on application launch, loading data from the server into the DataStore for quick use by the application (including creating, committing/storing, modifying, and deleting data records in the DataStore during runtime), and pushing all of the DataStore data back to the server at application close.

The DataStore also supports being able to load in data from on-disk "fixtures", which act independently from the server and allow for faster performance. During prototyping, a data model was created for a waveform and for a signal detection, including a relationship between them. One waveform could be related to many signal detections, and one signal detection could be related to a single waveform; this is seen in the SC.Record.toMany and SC.Record.toOne calls below.

```
NdcTest.WaveRecord = SC.Record.extend({
      index: SC.Record.attr(Number, {isRequired: YES}),
      waveformId: SC.Record.attr(Number, {isRequired: YES}),
      stationName: SC.Record.attr(String, {isRequired: YES}),
      signalDetections: SC.Record.toMany(
            'NdcTest.SignalDetection',
            {isMaster: YES, inverse: 'wave'}
      ),
});
```

Figure 7 – SproutCore: Waveform data representation

```
NdcTest.SignalDetection = SC.Record.extend({
      timestamp: SC.Record.attr(String),
      wave: SC.Record.toOne(
            'NdcTest.WaveRecord',
            {isMaster: NO}
      ),
});
```

Figure 8 – SproutCore: Signal detection data representation

3.1.2.1.2.4. Administration

SproutCore does not have a central administration system for distributing application updates or managing user access to those applications. SproutCore simply packages the application for delivery by compressing files. After that, it is up to the user to deploy the files to a web server.

3.1.2.1.2.5. Prototype

SproutCore prototyping used the D3 plotting package and created a waveform display and a signal detection table. These components communicated with each other such that updates to the signal detection table were received by the waveform display and vice versa. Due to time constraints, only one waveform is displayed in the final version. However, during the process of exploring the D3 plotting package, displays containing multiple waveforms were created.

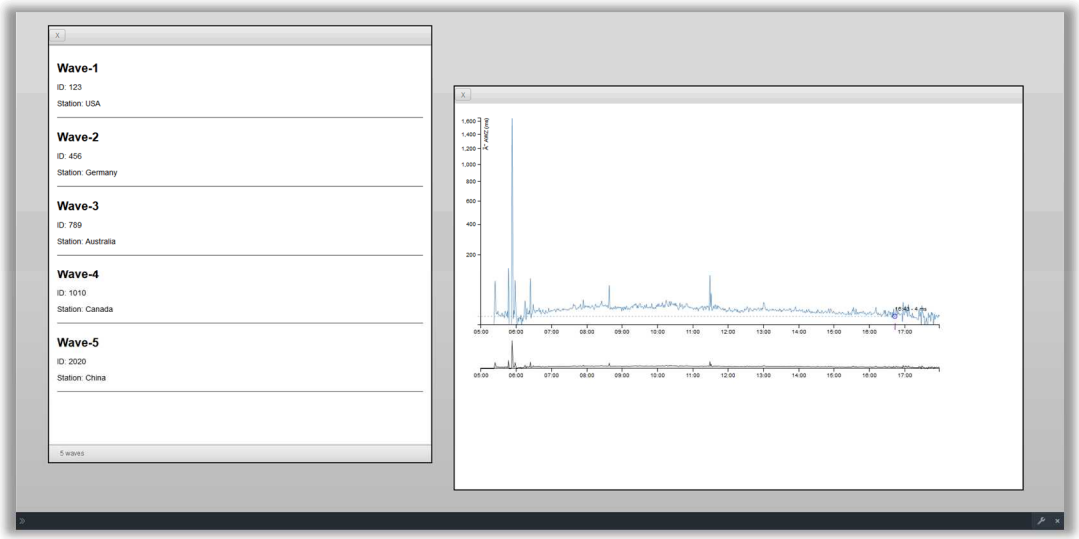

Figure 9 – SproutCore: Prototyping displays

3.1.2.1.2.6. Evaluation

While SproutCore had some useful features such as the KVO and DataStore, it was not quite the right fit for the prototyping goals. The fact that it does not support plug-in widget logic locks application developers into developing exclusively in SproutCore, desktop customization is cumbersome, it is difficult to use external JavaScript libraries, and SproutCore is implemented in a custom dialect of JavaScript. SproutCore is not a good fit for the system's user interface goals.

3.2. Eclipse 4 RCP

Many of the user interface requirements for the system relate to customization of the Analyst display. During E1, the UIF prototyping team found that for desktop applications, much of this functionality is immediately available when using a Rich Client Platform (RCP). RCPs limit the amount of developer time spent creating and maintaining window management and user customization code because that functionality is all provided and managed by the RCP.

The Eclipse 4 RCP is a strong runner up to the NetBeans RCP prototyped in E1. The JFace/SWT widget toolkit used by Eclipse has extensive market presence and developer communities. Eclipse is under active development. However, it does not have the extensive support options and documentation that Eclipse 3 has. IBM has contributed significantly to the Eclipse open source effort and continues to help drive its development.

The goal of the Eclipse prototyping was to compare its features related to display customization, development environment, performance, and display synchronization and communication with those provided by NetBeans RCP. This

was accomplished by creating a workspace with 3 display areas including a waveform display, a signal detection table, and a map. The waveform display should provide an interface for signal detection manipulation (adding/deleting/moving signal detections); the ability to zoom, pan, scroll; and display signal detection metadata via tooltips or a similar mechanism. The waveform display and signal detection table should be synchronized; changes to the signal detections on the waveform display should update the corresponding signal detection row in the signal detection table and vice versa.

3.2.1. Key Concepts

3.2.1.1. Dependency injection

Dependency injection is a software design pattern implementing inversion of control. In this paradigm, objects request services essentially by stating that they need them (e.g. by adding a framework object as a parameter to a constructor). In Eclipse 4 RCP, dependency injection is usually the only way to access underlying framework objects.

A benefit of dependency injection is that it loosens code coupling by late-binding dependencies to clients. However, a common complaint is that many errors are not found until runtime.

Pros	Cons
Loose coupling – less risk of problematic API change (i.e. static framework methods)	Hard (or impossible) to know if client will get framework object injected until runtime (code often compiles but fails at runtime)
Easier testing – clients are Plain Old Java Objects (POJOs), and their dependencies can be mocked up for unit testing	Access to framework objects is opaque – they "come from the sky"
Eliminates "ugly" chains of static method calls to get framework objects	Makes the API harder to learn – harder to use auto-complete in IDE to see what's available
	Framework objects can only be injected into classes that are directly linked to the Eclipse RCP UI element

Figure 10 – Eclipse RCP: Pros and cons of dependency injection with the Eclipse context

3.2.1.2. Dependency re-injection

The Eclipse framework uses dependency reinjection when it re-calls methods that take dependency injected objects when the objects have been changed in the context. This feature can act as a form of a listener in UI components for data model changes.

3.2.1.3. Eclipse context

The Eclipse context is a key-value store that the framework uses for dependency injection purposes, but it is also available to clients as a method of data sharing between objects. The context stores keys as Strings and the corresponding values as an instance of an object.

Parts (windows) and some other UI elements have their own Eclipse context, but it is often more useful to the use the top-level (global) Eclipse context. When a client requests an object via dependency injection, the framework engine looks in the current objects' Eclipse context and then up the hierarchy of contexts until a suitable object is found (or an exception is thrown when one isn't present). That top-level context where the search for objects to inject stops is often called the global context.

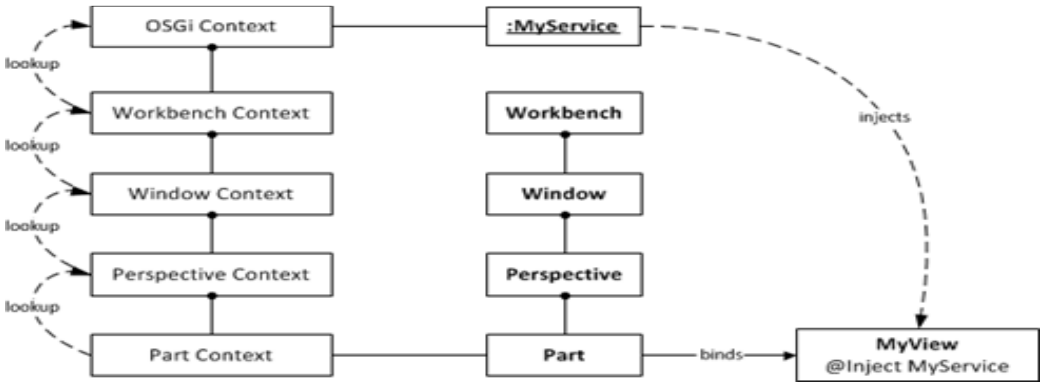

Figure 11 – Eclipse RCP: Dependency injection within Eclipse contexts

3.2.1.4. Application Model

The application model is the structure by which the Eclipse 4 RCP knows how to layout UI components.

At the surface, the application model editor provided by Eclipse RCP seems easy to use, but understanding the underlying semantics can be difficult. Adding or re-arranging UI components, adding menu items and toolbars, associating a control component with a class implementing the desired behavior, and setting up key bindings with this tool is pretty straightforward.

The application model can be edited through an Eclipse interface but the underlying .XMI (similar to XML) file itself can also be edited directly. Most of the application model can be changed at runtime by the client RCP application.

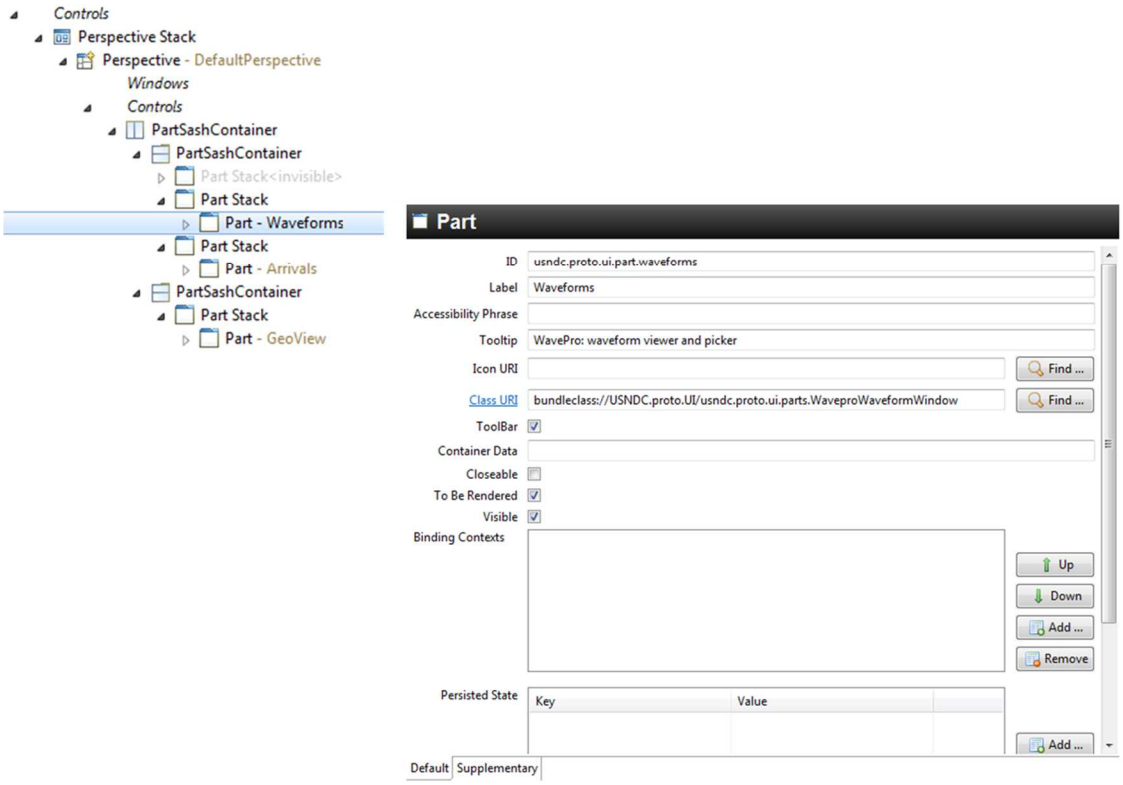

Figure 12 – Eclipse RCP: Editing the application model (for UI layout)

```
<?xml version="1.0" encoding="UTF-8"?>
<application:Application xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:advanced="http://www.eclipse.org/ui/2010/UIModel/application/ui
 <children xsi:type="basic:TrimmedWindow" xmi:id="_9ockY0s2EeOS7OhktwCL0g" label="USNDC proto UI" width="500" height="400">
  <tags>shellMaximized</tags>
  <children xsi:type="advanced:PerspectiveStack" xmi:id="_9ockf-s2EeOS7OhktwCL0g" elementId="MainPerspectiveStack">
   <children xsi:type="advanced:Perspective" xmi:id="_9ockg0s2EeOS7OhktwCL0g" elementId="usndc.proto.defaultperspective" label="DefaultPerspective">
    <children xsi:type="basic:PartSashContainer" xmi:id="_9ockges2EeOS7OhktwCL0g" elementId="" horizontal="true">
     <children xsi:type="basic:PartSashContainer" xmi:id="_b_Z-gPV5EeOjMr9NyVOplw" elementId="usndc.proto.ui.partsashcontainer.2" containerData="">
      <children xsi:type="basic:PartStack" xmi:id="_eB65MAFlEeSgSch0J_M3tA" elementId="usndc.proto.ui.partstack.fkPartStack" visible="false">
       <children xsi:type="basic:Part" xmi:id="_enS3lAFlEeSgSch0J_M3tA" elementId="usndc.proto.ui.part.fkPart" contributionURI="bundleclass://USNDC.proto.UI/usndc.proto.ui.fkPlot.FKPlotter" label="FKPlot"/>
      </children>
      <children xsi:type="basic:PartStack" xmi:id="_ijXmYPV5EeOjMr9NyVOplw" elementId="usndc.proto.ui.partstack.waveforms">
       <children xsi:type="basic:Part" xmi:id="_cqQjsOtEEeOKquXcqjPpHQ" elementId="usndc.proto.ui.part.waveforms" contributionURI="bundleclass://USNDC.proto.UI/usndc.proto.ui.parts.WaveproWaveformWindow" la
        <toolbar xmi:id="_hHgYkAk0EeSsfPcTe37dTg" elementId="usndc.proto.ui.toolbar.pickMode">
         <children xsi:type="menu:HandledToolItem" xmi:id="_jgeWoAk0EeSsfPcTe37dTg" elementId="usndc.proto.ui.handledtoolitem.P" label="Pick mode: P" tooltip="Pick on P (Alt+P)" type="Radio" command="_A_KN0Al
         <children xsi:type="menu:HandledToolItem" xmi:id="_uv_VkAk2EeSsfPcTe37dTg" elementId="usndc.proto.ui.handledtoolitem.S" label="Pick mode: S" tooltip="Pick on S (Alt+S)" type="Radio" command="_A_KN0Ak
         <children xsi:type="menu:HandledToolItem" xmi:id="_kZQD4Ak5EeSsfPcTe37dTg" elementId="usndc.proto.ui.handledtoolitem.PKP" label="Pick mode: PKP" tooltip="Pick on PKP (Alt+K)" type="Radio" command="
         <children xsi:type="menu:HandledToolItem" xmi:id="_fMKy8Bj7EeSFEYK0QF3OtA" elementId="usndc.proto.ui.handledtoolitem.DELETE" label="Pick mode: Delete" type="Radio" command="_A_KN0AkNEeSsfPcTe3
        </toolbar>
       </children>
      </children>
      <children xsi:type="basic:PartStack" xmi:id="_LK81cAMEEeSTdqHgtAgU9A" elementId="usndc.proto.ui.partstack.arrivalStack">
       <children xsi:type="basic:Part" xmi:id="_LjLIMAMEEeSTdqHgtAgU9A" elementId="usndc.proto.ui.part.0" contributionURI="bundleclass://USNDC.proto.UI/usndc.proto.ui.parts.WaveproArrivalWindow" label="Arriva
      </children>
     </children>
     <children xsi:type="basic:PartSashContainer" xmi:id="_ooVaAA0iEeSDiLZd2LOo7Q" elementId="usndc.proto.ui.partsashcontainer.1">
```

Figure 13 – Eclipse RCP: XMI representation of the application model (snippet)

3.2.2. SWT / JFace toolkit

The Eclipse UI toolkit is built on the Standard Widget Toolkit (SWT) [5] with the JFace [6] toolkit. SWT uses a Java Native Interface (JNI) to windowing API's of each particular operating system it runs on. Due to this, it tends to be faster than

alternative Java widget toolkits like Swing/AWT. It is unclear how significant the performance difference is between SWT/JFace and Swing/AWT.

SWT was originally developed by IBM and is maintained by the Eclipse Foundation (with IBM support). Eclipse is the largest and most well-known product that runs on SWT; other IBM products such as the IBM Rational Software suite also use SWT. It is unclear whether IBM and/or Eclipse will be supporting SWT indefinitely.

Pros	Cons
Fast and responsive, by using JNI to OS windowing API	Not truly platform-independent; requires libraries to run on a given platform
Native look and feel	API somewhat hard to use
Swing widgets can be embedded into SWT widgets (not vice versa currently)	Few customization options; SWT classes cannot be extended by clients
	Client code must explicitly dispose some resources to avoid leakage

Figure 14 – Eclipse RCP: SWT Pros and Cons

The role of JFace is to make some common UI programming tasks using SWT easier and less tedious. The last stable release was in July of 2013, and it is developed and maintained by the Eclipse Foundation. JFace is dependent on SWT, but SWT is not dependent on JFace – developers can use JFace if they find it convenient, but anything that can be done in JFace can be done directly in SWT. As such, the future of JFace completely relies upon the success and adoption of SWT.

3.2.3. JavaFX / JavaFX 2.0 integration

It is possible to use JavaFX widgets in an Eclipse RCP 4 application. This requires a class provided as part of "e(fx)clipse" project [7]. This class, FXCanvas, can take a SWT Composite object as input and has a method setScene() that takes a JavaFX scene. Once configured, a Java FX Scene can be added to an Eclipse 4 RCP UI component as follows:

```
@PostConstruct
void init(Composite parent) {
      FXCanvas canvas = new FXCanvas(parent, SWT.None);
      final Group rootGroup = new Group();
      final Scene scene =
            new Scene(rootGroup, 800, 400, Color.THISTLE);
      // customize scene as desired
      canvas.setScene(scene);
}
```

Figure 15 – Eclipse RCP: Adding a JavaFX scene to an Eclipse RCP part

The class containing this method needs to be associated with an Eclipse RCP part, which will call this method after being built, passing in the SWT Composite object that implements the part.

3.2.4. Prototype

The Eclipse RCP prototype utilized WavePro, a Sandia-developed waveform plotting package, for the waveform display and signal detection table portions of the prototype. Integrating WavePro with Eclipse required minor modifications to allow the wrapper Eclipse view classes to grab the relevant items from WavePro.

WorldWind [8], as tailored for the Sandia GeoView project, was used for the map display portion of the prototype. This required adding a variety of dependencies to the application, which was not straightforward.

While researching the best way to utilize existing libraries within the Eclipse RCP, it was discovered that code running outside of the RCP application project runs with different Java class loaders and threads [9]. This leads to runtime problems if outside projects attempt to access RCP services. This aspect of the Eclipse RCP is not included in the Eclipse documentation. Instead, the prototyping team discovered this feature when the Eclipse plugin for the WavePro code could not access Eclipse RCP services, it could not be updated via dependency injection, and static Eclipse 4 RCP methods are unavailable.

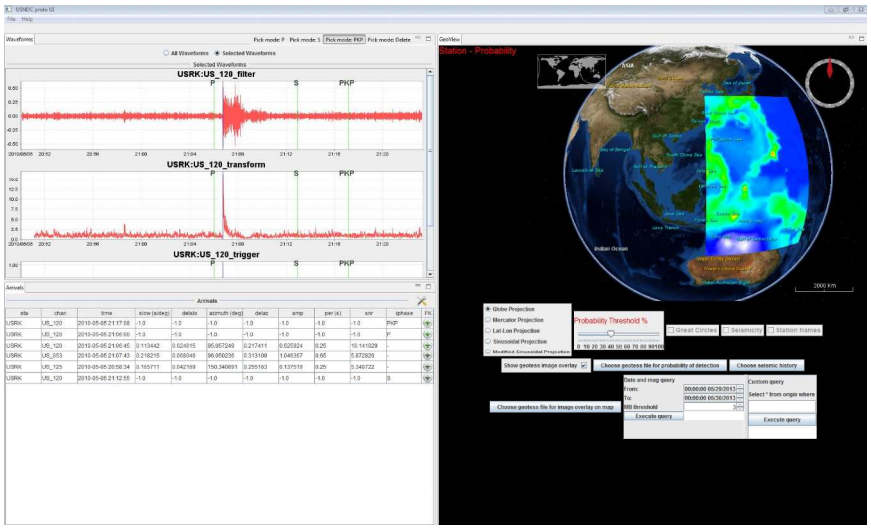

Figure 16 – Eclipse RCP: Prototype

3.2.4.1. Communication

The waveform display and the signal detection table communicate using dependency re-injection and the global Eclipse context. If either the waveform display or the signal detection table change the detections list in the Eclipse context, this causes the waveform and table displays to execute the necessary code to update their displays accordingly. This is an example of the Eclipse context acting as a publish/subscribe system.

```
@Inject @Optional
public void pickChangedInArrivalsPanel
      (@Named(DIUtility.picksMap) Map<String, Arrival> picksMap)
{
      // update waveform displays
```

Figure 17 – Eclipse RCP: Using the Eclipse context as a listener for data changes

In the code example above, the "picksMap" variable is added to the global Eclipse context when the application starts. The Java Map stored there has been updated by a user editing an entry in the Arrival table. Upon that change, this method is called with the new value of the Map, allowing the waveform plot to update accordingly. The @Named annotation refers to the key that the given value was associated with in the Eclipse context, and the @Inject annotation indicates that the arguments of this method should be obtained through dependency injection. It is very important to note that a method as above expecting to use dependency injection will only work if the class it is located in is directly associated with an Eclipse part in the application model.

Eclipse also offers a way for methods to become listeners by being annotated to state that they are listening for a particular event String (a "UI topic"), and events can be published with a call like IEventBroker.send("the topic"). All methods with the relevant UI topic annotation will be called. An object can also be passed through this mechanism that can be used by listening methods. This event system is really just an alternative to dependency injection. It doesn't seem to provide additional functionality beyond that.

3.2.4.2. User Preferences

One of the advantages of using an RCP is the ability for users to customize the layout of the tabbed workspace windows in the workspace. Perspectives can be created that define the initial contents and layout of a workspace, but Eclipse is set up to remember user placement of these windows when the application is closed and reopened. Users can resize, nest, undock, dock, and otherwise independently manipulate each of the parts within the main application window.

Hot keys can be mapped to the desired action through the Eclipse framework. In the prototype, the picking mode/phase is set either using buttons in the toolbar of the waveforms view or hot keys, such as "Alt-S" for changing to S phase.

The UI color scheme supports CSS for styling, which provides the flexibility to customize user based on individual user color preferences.

3.2.5. Evaluation

Internet opinions regarding Eclipse 4 RCP are divided: developers either passionately advocate for the use of the Eclipse RCP for UI development or vehemently warn developers to steer clear. Common complaints included a lack of documentation and tutorials and opaque methods of obtaining framework objects. Touted benefits of the framework are the use of dependency injection as a way of loosening code coupling, cross-platform capability, large community for support (IBM and The Eclipse Foundation), and a GUI for editing and re-arranging UI elements for the application. The future of Eclipse 4 RCP seems uncertain, as it has been out for over 4 years at the time of this writing, but has little online documentation or evidence of a strong user base (both of which Eclipse 3.x did have). Currently, the primary resource for beginning Eclipse 4 RCP developers is a tutorial by Lars Vogel [10].

4. CONCLUSIONS

It was valuable to learn about different web application development frameworks and to become more knowledgeable about where this area of user interface development is headed. While browser based user interfaces might satisfy many of the system requirements (unified UIF; consistent, modern interface; responsive and customizable interface; use of COTS), some concerns were highlighted during this prototyping effort. One of the UIF requirements is the support of a modular, component-based architecture; this may be possible using browser based UI frameworks, but it is not something enforced by those frameworks. A related concern is integration with underlying mechanisms of the architecture (e.g. the Object Storage and Distribution mechanism); these browser based frameworks may not interface well with the underlying architecture, and the languages used for within a browser based UI (e.g. JavaScript) will not be the same as those chosen for the overall architecture. For these reasons, the UIF prototyping team concluded that an RCP solution better meets Analyst needs.

There are requirements related to non-Analyst user interfaces (e.g. system monitoring and analysis) that could be met with a browser based display. In cases where the user benefits from viewing multiple web pages simultaneously, a framework such as OWF could provide a nice dashboard to facilitate management of those pages.

The Eclipse RCP is a powerful tool that relieves developers of the need to create and maintain a full suite of windows management and user preferences code. However, it is uncertain what level of support will be provided for the Eclipse

RCP, SWT, and JFace in the future, and it is unclear how well the upcoming Java UI toolkits such as JavaFX will truly integrate with the underlying framework.

5. PATH FORWARD

Prototyping results indicate NetBeans / JavaFX RCP is the most viable OSS candidate for the UIF and the SNL team has selected it for use in executable architecture development.

The path forward for the UIF prototyping team includes more extensive prototyping of potential solutions for the non-Analyst user interfaces. The non-Analyst user interface requirements could be met with either NetBeans or a web-based framework such as the Ozone Widget Framework.

Even though the frameworks for the executable architecture development have been chosen, surveys will still be conducted as needed to evaluate which toolkits, libraries, or embedded software packages will be needed.

6. WORKS CITED

[1] Sandia National Laboratories, "US NDC Modernization Iteration E1 Prototyping Report: User Interface Framework".

[2] "What is Highstock?," Highcharts, 2014. [Online]. Available: http://www.highcharts.com/products/highstock.

[3] "Ozone Widget Framework," Next Century Corporation, 2014. [Online]. Available: http://www.ozoneplatform.org/.

[4] "SproutCore," SproutCore, [Online]. Available: http://sproutcore.com/.

[5] "SWT: The Standard Widget Toolkit," The Eclipse Foundation, 2014. [Online]. Available: http://www.eclipse.org/swt/.

[6] "JFace," Wikipedia, 2014. [Online]. Available: http://en.wikipedia.org/wiki/JFace.

[7] "e(fx)clipse JavaFX Tooling and Runtime for Eclipse and OSGi," The Eclipse Foundation, [Online]. Available: http://www.eclipse.org/efxclipse/index.html.

[8] "NASA World Wind," NASA, 2011. [Online]. Available: http://worldwind.arc.nasa.gov/.

[9] "Eclipse - a tale of two VMs (and many classloaders)," Eclipse Zone, [Online]. Available: http://www.eclipsezone.com/articles/eclipse-vms/.

[10] L. Vogel, "Eclipse 4 RCP - Tutorial," Vogella, 2013. [Online]. Available: http://www.vogella.com/tutorials/EclipseRCP/article.html.

7. APPENDIX A. SURVEY OF PLOTTING PACKAGES FOR WAVEFORM DISPLAY

The following table lists the plotting packages considered for the waveform display included in the browser based UI prototyping.

Table 1 – Summary of survey of plotting packages

Candidate Plotting Solution	Description	Assessment
Data-Driven Documents (D3) Data-based document manipulation http://d3js.org/	D3 is a JavaScript library which facilitates integration with any web page. It is framework independent and should work with OWF and SproutCore. D3 claims to have a responsive interface that performs well with large amounts of data and individual waveforms can be dragged and dropped within the plot area.	Prototyping with D3 would allow the team to learn more about using external JavaScript libraries within web frameworks. D3 supports multiple waveforms and claims to have a responsive interface. **Prototype?** Yes
Highcharts: Highstock Interactive JavaScript charts http://www.highcharts.com/	Highstock is a JavaScript library which facilitates integration with any web page. It is framework independent and should work with OWF and SproutCore. Highstock supports annotations of plotted data which can be used to represent signal detections along the plotted waveforms.	The prototyping team decided to prototype this software package since Highcharts seems to be growing in popularity. **Prototype?** Yes
Tableau http://www.tableausoftware.com/	Tableau's mission is to help users understand their data. This software creates beautiful plots, but it was not available for free. In addition, it seems to be a standalone application that may not integrate well with other applications.	Since Tableau's goal is to provide data display functionality with minimal customization, it did not seem like a good fit for prototyping efforts that wanted more access to customization of the data displays.

		Prototype? Not now, perhaps in the future
Dygraphs JavaScript charting library http://dygraphs.com/	Dygraphs has many of the same features as D3 and Highcharts but seems more dated and less modern.	Because D3 and Highcharts seem to have all of the features of Dygraphs while being under more active development, Dygraph was not chosen for prototyping. **Prototype?** Not now, perhaps in the future
SoundCloud https://developers.soundcloud.com/	While SoundCloud has some plotting capabilities related to displaying music files, it is focused on building applications for sharing audio across the web.	SoundCloud is too heavily focused on audio files to be useful for prototyping. **Prototype?** No
WaveSurfer.js Customizable waveform audio visualization	Similar to SoundCloud, this library is tailored to displaying and manipulating music files.	Wavesurfer is too heavily focused on audio files to be useful for prototyping. **Prototype?** No

8. APPENDIX B. SURVEY OF BROWSER BASED UI FRAMEWORKS

The following table lists the browser based UI framework packages considered for the browser based UI prototyping.

Table 2 - Summary of survey of browser based UI frameworks

Candidate Framework	Description	Assessment
	Widget Framework	
Ozone Widget Framework http://www.ozoneplatform.org/	OWF provides workspaces for widgets. Each widget is a web page, but communication can take place between widgets. Dashboards or workspaces are layouts used to arrange and display your widgets.	OWF's capability to provide independent web pages encapsulated in widgets, customizable desktops for the layout of those widgets, and communication among the widgets seems to provide a lot of the functionality needed in the Analyst workspace. **Prototype?** Yes
	Plugin	
JavaFX (as a browser plugin) http://www.oracle.com/technetwork/java/javafx/overview/index.html http://docs.oracle.com/javafx/2/deployment/deployment_toolkit.htm	The Java/Oracle user interface development code base is migrating from Swing to JavaFX. In addition to creating desktop applications, JavaFX is a software platform for creating and delivering rich internet applications (RIAs) that can run across a wide variety of devices.	It is worth exploring if JavaFX would support an Analyst interface from either a browser or an RCP. **Prototype?** Investigate how JavaFX integrates with the Eclipse RCP
	Web Application Framework	
SproutCore http://sproutcore.com/	SproutCore is a framework for building native-caliber web applications and introduces its own language and syntax to support object-oriented programming that extends JavaScript objects. At runtime, native language files are converted into HTML and rendered to a web	SproutCore offers a "workspace view" that can contain multiple pages that the user can arrange as they wish. In addition, the prototyping team wanted to become familiar with a web framework that offered more than what

	page. SproutCore pushes the entire MVC design pattern into the browser, with the only separate component being the data that is dynamically loaded into the model during application runtime.	is standard with JavaScript only pages. For these reasons, SproutCore was considered a candidate for further prototyping. **Prototype?** Yes
Ruby on Rails http://rubyonrails.org/	Open source web application framework which runs via the Ruby programming language. It is a full-stack framework: it allows creating pages and applications that gather information from the web server, talk to or query the database, and render templates out of the box. Ruby is a dynamic, reflective, object-oriented, general-purpose programming language. It supports multiple programming paradigms, including functional, object-oriented, and imperative. It also has a dynamic type system and automatic memory management.	Ruby on Rails appears to be a powerful web development language. Due to time constraints, this was not prototyped. **Prototype?** No
qooxdoo http://qooxdoo.org/	Universal JavaScript framework with a coherent set of individual components and a powerful tool chain. With qooxdoo you build rich, interactive applications, native-like apps for mobile devices, light-weight single-page oriented web applications or even applications to run outside the browser.	qooxdoo appears to be a powerful web development framework. Due to time constraints, and the fact that this JavaScript only, this was not prototyped. **Prototype?** No
Ember.js http://emberjs.com/	Ember.js evolved out of SproutCore and is described as "a JavaScript framework for creating ambitious web applications that eliminates boilerplate and provides a standard	Because Ember.js developed out of SproutCore, and the decision had already been made to prototype SproutCore, the team decided not to

	application architecture." It comes tightly integrated with a templating engine known as Handlebars, which gives Ember one of its most powerful features: two-way data-binding. JavaScript is a flexible and powerful language but it also has its shortcomings. Out of the box it doesn't offer the sort of functionality that lends itself to MVC style development. So Ember has extended the base.	prototype Ember.js this iteration. It appears to be a powerful web application framework that adds to the capabilities provided by JavaScript and SproutCore. **Prototype?** No
	WebUI Framework / Toolkit	
Sencha http://www.sencha.com *Sencha Ext JS* http://www.sencha.com/products/extjs/ *Sencha GXT* http://www.sencha/com/products/gxt	Sencha is an open source web application framework that creates development frameworks and tools that help design, develop, and deploy applications for desktops and mobile devices. Sencha Ext JS is a JavaScript Framework for Rich Desktop Apps and provides a desktop application development platform Sencha GXT is an Application Framework for Google Web Toolkit. It uses the Google Web Toolkit compiler so developers can write applications in Java and compile code into highly optimized cross-browser HTML5 and JavaScript.	The fee for Sencha licenses seemed too high for an exploratory prototyping effort. However, should the US NDC decide to invest heavily in browser based user interfaces, Sencha is a framework that definitely merits further investigation. It provides many different toolkits and supports a wide range of applications, including “desktop like” web applications. **Prototype?** No
Cappuccino http://www.cappuccino-project.org/	Open source framework that makes it easy to build desktop-caliber applications that run in a web browser. Uses Objective-J, which is modelled after Objective-C and built entirely on top of JavaScript.	Due to time constraints and the requirement to learn a new programming language (Objective-J), this framework was not chosen for prototyping. **Prototype?** No

	Portal / Portlet	
Liferay https://www.liferay.com/	Provides a unified web interface for data, tools, and system integrations scattered across a large number of resources and devices. Within the portal, the portal web page interface is composed of a number of Portlets. These portlets are self-contained interactive elements written to a particular standard, such as JSR 168 or JSR 286. As portlets are built independent of the portal and are loosely coupled with the portal, they are apparently built using SOA.	Liferay is more than just a user interface framework; it includes the entire underlying SOA architecture. Because this prototyping effort was focused on the user interface framework while other prototyping efforts were focused on the underlying system services, Liferay was not a good fit for the UIF prototyping. **Prototype?** No

This is the last page of the document.