Voice Foundation Classes

The Unified CVP Voice Foundation Classes are a Java API for generating VoiceXML. Any custom component wishing to produce VoiceXML must use the VFCs because their main purpose is to act as an abstraction layer between VoiceXML and the component. The VFCs handle the vagaries of VoiceXML and especially the differences in the VoiceXML interpreted by various voice browsers. This allows the developer to simply focus on the functionality desired without worrying about the details of writing VoiceXML or the quirks of their chosen voice browser. The VFCs are primarily used to construct voice elements, though hotevents and on call end classes use the VFCs as well.

VFC Design

The high level design of the VFCs is to simulate standard VoiceXML in Java. The behavior of these classes directly matches the VoiceXML specifications (both versions 1 and 2). This, however, acts only as a basis from which supporting a particular voice browser begins, since no two browsers have exactly the same compliance. The software provides voice browser compatibility by extending these base VFCs to create a layer that produces the VoiceXML compatible with a particular voice browser. Most of the functionality is still defined in the base VFC classes and only the browser-specific functionality needs to be included in the subclasses. The classes for a particular voice browser are encapsulated in a separate plugin or driver, called a Gateway Adapter. Installing a new Gateway Adapter will add support for a new voice browser and a Unified CVP application can be deployed on a new browser by simply selecting the Gateway Adapter to use.

The design of the base VFCs follows roughly the design of VoiceXML, utilizing similar concepts and naming, so prior knowledge of VoiceXML is beneficial for understanding the VFC design. The VFCs allow full compatibility with VoiceXML in that anything you can do in VoiceXML you can do in the VFCs, including using proprietary tags and/or attributes introduced by supported browsers.

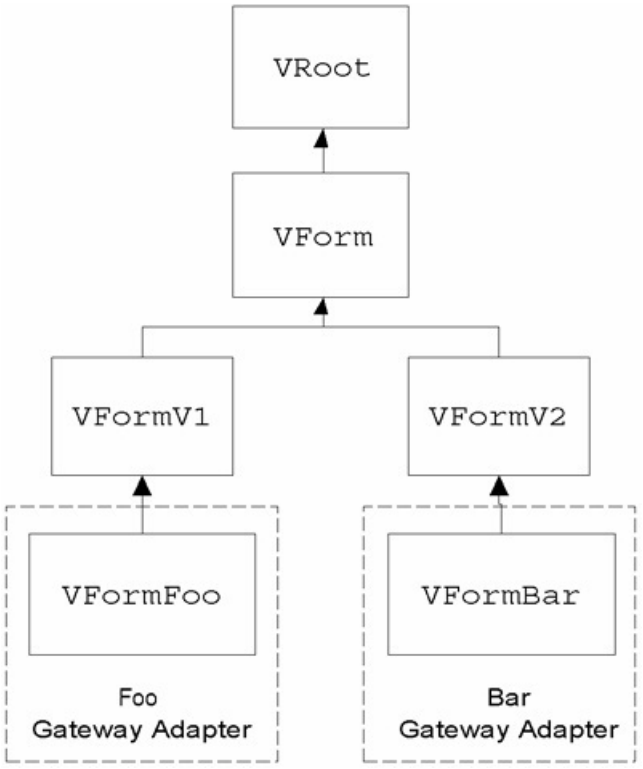
Many times, a single VoiceXML tag maps to a single VFC that is similarly named. The class `VForm`, for example, deals with VoiceXML `<form>` tags and the class `VField` with `<field>` tags. Some tags, however, have been combined into a single VFC for ease of use. For example, the `VAction` class encapsulates tags from `<var>` and `<assign>` to `<break>` and `<submit>`. As a result, there are fewer VFCs than VoiceXML tags. The VFCs also help the developer by producing some VoiceXML automatically. The developer will quickly find that using the VFCs is very much like coding in VoiceXML, except in Java.

There are a few concepts that need to be described before delving into the individual VFCs. First, each VFC class extends a common base class, `VRoot`. The purpose for this is similar to having all Java classes extend

`Object`, it is a way to help define common functionality of the VFCs as well as being able to identify if a Java class is a VFC.

The second concept involves the hierarchy of the VFCs. There are, in fact, several layers of abstraction in the VFCs that separate not only differences between various voice browsers but also versions of the VoiceXML standard. There are separate VFCs for VoiceXML version 1.0 and version 2.0, and the similarities are encapsulated in a common base class. The following figure, shows this graphically with the `VForm` class. The main `VForm` class extends the `VRoot` class and is itself extended by `VFormV1` and `VFormV2`, representing VoiceXML 1.0 and 2.0 compliances. Luckily, there are only a few differences between these versions, so the developer can still do most coding to the base `VForm` class. The Gateway Adapters introduce VFCs that extend `VFormV1` or `VFormV2` depending on whether the voice browser it supports is compatible with VoiceXML 1.0 or 2.0.

Figure 1: VForm and Its Extensions



The last concept is that VFC objects are not instantiated using the `new` keyword. A static factory method named `getNew` is used instead. The reason for this is related to the abstraction of the voice browser differences. As mentioned previously, a developer need only code using the base VFC classes. At runtime, the factory methods used to instantiate VFC classes actually returns the appropriate voice browser-specific VFC (for example, `VFormFoo` in the illustration). But since the developer treats the return object as the base VFC, that object is downcasted. This is the heart of the VFC abstraction design. Since all VFC derivative classes extend their base VFCs, a developer need only code to the base VFCs and that automatically makes their code compatible with any voice browser represented by a Gateway Adapter.

In order to identify which voice browser VFC to return, every factory method must include as its first argument an instance of `VPreference`. `VPreference`, while a VFC class, does not match to a VoiceXML tag, it is used instead to hold preferences made by the user in Builder for Call Studio for the application, such as the voice browser and default audio path. By passing this object to all factory methods, the appropriate object can be

returned. The `VPreference` instance is automatically created for the developer and made available through the Session API passed to voice elements, hotevents, or call end classes.

The following Java code demonstrates the concepts described above:

```
VPreference pref = ved.getPreference();
VForm form = VForm.getNew(pref, "start");
```

Here, the `VPreference` object is obtained from the `VoiceElementData` object passed as input to a voice element. It is used to create a `VForm` object. Assuming the application is using the voice browser Foo, that choice is reflected in the `VPreference` object and therefore the `getNew` factory method returns a `VFormFoo` object, which is automatically downcasted to a `VForm` object. The developer then uses the form object as desired.

This ability to treat all objects returned as a root VFC object is not available when the developer wishes to use functionality that exists either in a particular version of VoiceXML or a particular voice browser. The developer must understand that doing so would prevent their code from functioning on all voice browsers. In this case, the developer simply treats the return of the factory method as a class higher in the class hierarchy (`VFormV2` or `VFormFoo` in the illustration).

The following Java code demonstrates this:

```
VGrammar myGrammar = VGrammar.getNew(pref);
((VGrammarV2) myGrammar).setMaxage(1000);
```

The `setMaxage` method exists only in the `VGrammarV2` class since this is a feature that exists only in VoiceXML 2.0. To call this method, one must first upcast the previously downcasted object back to `VGrammarV2`. If this is not done, an exception will be thrown indicating that `VGrammar` does not have a method named `setMaxage`.

Note If the user in the Builder chose a voice browser that was compatible with VoiceXML 1.0 only, a runtime exception would be thrown when this code is encountered because that browser would be unable to understand VoiceXML referring to maxage.

VFC Classes

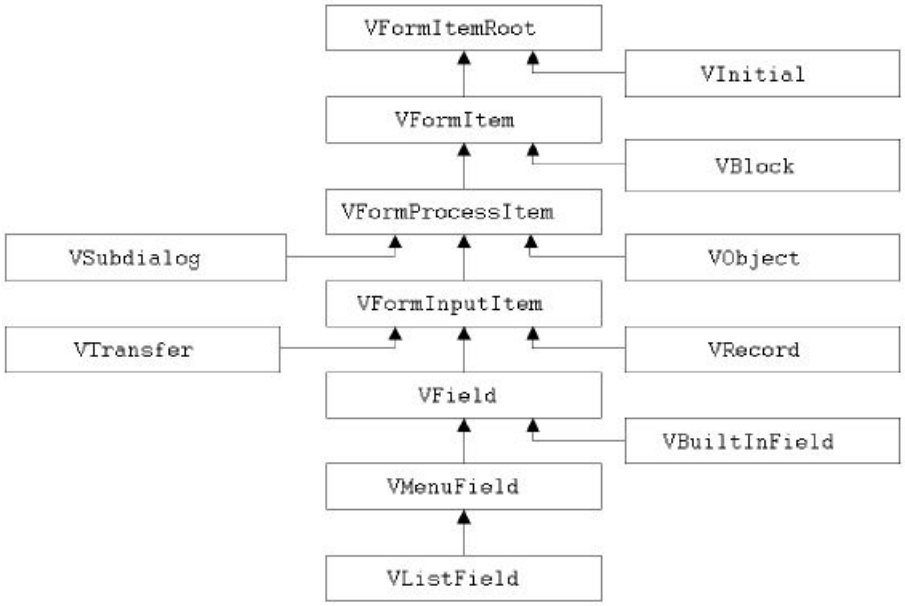
The following lists all the VFC classes (with full package names) and briefly explains what they are responsible for. The Javadocs for the VFCs provide significantly more detail about the classes, their methods, and how they are used.

- **com.audium.core.vfc.util.VMain** – This object is the container for a complete VoiceXML document. It includes methods for managing information about the page such as the meta tags, the doc type, and the value to put in the `<vxml>` tag's `xml:lang` attribute. It includes methods for adding document-scope data such as links, variables, and VoiceXML properties. `VForm` objects are added to this object to create the VoiceXML page. VXML Server uses the `VMain` object to handle the printing of the VoiceXML page. Voice elements receive an instantiated `VMain` object as input and the developer need only worry about filling the object with the appropriate content.
- **com.audium.core.vfc.form.VForm** – This class is a container for all the content in a VoiceXML page not handled by the `VMain` class. It is a direct mapping of the `<form>` tag, though it also produces other form-level tags such as `<var>` or `<filled>`.
- **com.audium.core.vfc.list.VList** – This class is used to encapsulate a list of items that can be deployed as either a traverse list or a streaming list. A traverse list presents a menu after an item is presented that allows the caller to move forwards and backwards through the list. A streaming list is one where all the

items are played one after the other. This VFC class does not reflect any VoiceXML tags, it was produced by Unified CVP to facilitate the creation of lists within VoiceXML. The class outputs a set of forms that implement the list.

- **Form Items** – The `VForm` classes encapsulate most of the content of a VoiceXML page, and each form has any number of form items added to it. These form items span the range of capturing input from the caller to performing a telephony transfer. Each form item has a different purpose, though many form items share features in common. The VFCs relate the classes that handle each form item by creating a hierarchy starting with the simplest form items, with features common to all, to more complex form items that add features through each successive class extension. The following figure shows this class hierarchy and a description of each branch is listed after the figure.

Figure 2: VForm Class Hierarchy



VForm Hierarchy branches and descriptions:

- **com.audium.core.vfc.form.VFormItemRoot** – This class is the base class for all form items. It defines the ability to include audio, which every form item shares.

  com.audium.core.vfc.form.VInitial – This class is used when performing mixed initiative data capture. Mixed initiative data capture is a way of capturing multiple inputs in one utterance, such as a person's first and last names together rather than having to prompt for each individually. It is a direct mapping of the `<initial>` tag.

- **com.audium.core.vfc.form.VFormItem** – This class defines a standard form item. It defines the ability to perform actions within the form item (some form items do this within a `<filled>` tag).

  com.audium.core.vfc.form.VBlock – This class deals with producing a block in which any action and/or audio can be placed. It is a direct mapping of the `<block>` tag.

- **com.audium.core.vfc.form.VFormProcessItem** – This class defines form items that process data from the caller or an external source. Process form items define the ability to catch and handle VoiceXML events and refer to VoiceXML properties.

com.audium.core.vfc.form.VSubdialog – This class is used to make a call to a VoiceXML subdialog. A subdialog acts very much like a function call in VoiceXML, performing some encapsulated task and then returning to the calling context. It is a direct mapping of the `<subdialog>` tag.

com.audium.core.vfc.form.VObject – This class is used to produce VoiceXML that calls an external data object. It is a direct mapping of the `<object>` tag.

- **com.audium.core.vfc.form.VFormInputItem** – This class defines form items that take input from the caller. Input form items define a grammar to capture the data.

  com.audium.core.vfc.call.VTransfer – This class deals with performing a telephony transfer. It is a direct mapping of the `<transfer>` tag.

  Note The reason this is considered an input form item is because theoretically according to the VoiceXML specification, a grammar can be active within a call transfer. This, though, is rarely used or supported.

  com.audium.core.vfc.audio.VRecord – This class deals with performing a recording of the caller's voice. It is a direct mapping of the `<record>` tag.

- **com.audium.core.vfc.form.VField** – This class defines field form items, which deal with capturing utterances from the caller and converting them into information. Fields define the ability to specify utterance links.

  com.audium.core.vfc.form.VBuiltInField – This class deals with producing fields that capture data specified by grammars built into the voice browser. Any voice browser supporting VoiceXML is required to support data capture of numbers, dates, times, currency values, phone numbers, digit-by-digit values, and boolean values (yes / no). The class produces `<field>` tags as well as other field-related tags such as `<prompt>` and `<filled>`.

  - **com.audium.core.vfc.form.VMenuField** – This class is used when a menu is desired in the VoiceXML document. The class produces `<field>` tags with `<option>` tags defining each menu option. The `<menu>` and `<choice>` tags in VoiceXML are just shortcuts for this and cannot be produced with the VFCs.

  - **com.audium.core.vfc.form.VListField** – This class is a special kind of menu that is used by the `VList` class when it is configured to act as a traverse list. The menu is pre-built to support options to go forwards and backwards.

- **com.audium.core.vfc.util.VAction** – This VFC class encapsulates multiple VoiceXML tags that represent taking certain actions. All the VoiceXML tags produced by this class have the same parent tags and so can be used in the same locations. Combining these tags into one class reduces the complexity of the VFCs since special handlers are not needed for each tag. The following lists the actions that the `VAction` tag encapsulates and the corresponding VoiceXML tag: variable declarations (`<var>`), variable assignment (`<assign>`), gotos (`<goto>`), HTTP submits (`<submit>`), clearing forms and fields (`<clear>`), scripts (`<script>`), logging (`<log>`), throwing events (`<throw>`), reprompting (`<reprompt>`), returning from subdialogs (`<return>`), disconnects (`<disconnect>`) and exits (`<exit>`).

- **com.audium.core.vfc.audio.VAudio** – This class deals with audio, both TTS and through audio files. A single `VAudio` object can contain any number of audio items (so an entire voice element audio group can be encapsulated in one `VAudio` object). The class also deals with playing back a recording, managing bargein, adding pauses to the playback.

Note SSML (Speech Synthesis Markup Language) that is entered by the application designer in Builder for Call Studio is handled correctly by this class.

- **com.audium.core.vfc.util.VEvent** – This class handles VoiceXML events and what to do when they occur. Events may be user-triggered such as nomatch or noinput events, or custom events thrown by the developer or voice browser. Hotevents are basically `VEvent` classes that VXML Server adds to the VoiceXML root document. It is a direct mapping of the `<catch>` tag.

Note The `<noinput>`, `<nomatch>`, and `<help>` tags are all shortcuts for variations of the `<catch>` tag so are not produced by the VFCs.

- **com.audium.core.vfc.util.VGrammar** – This class deals with specifying either an inline or external DTMF or speech grammar. It is a direct mapping of the `<grammar>` tag.
- **com.audium.core.vfc.util.VIfGroup** – This class deals with producing an if statement within VoiceXML. It is a direct mapping of the `<if>`, `<elseif>`, and `<else>` tags.
- **com.audium.core.vfc.util.VLink** – This class deals with creating an utterance-activated link within the VoiceXML page. It is a direct mapping of the `<link>` tag.
- **com.audium.core.vfc.util.VProperty** – This class deals with including VoiceXML properties in the VoiceXML page. It is a direct mapping of the `<property>` tag.
- **com.audium.core.vfc.VException** – This exception class is thrown when a VFC class encounters an error.
- **Utility Classes** – These classes are used by the VFCs to aid in the organization of data they require. The following lists those classes:
  - **com.audium.core.vfc.util.VoiceInput** – This class is used to encapsulate how input is to be expected from the caller. It can encapsulate voice only input, DTMF only input, or both. It is also used to specify what data to look for, which can be a single or multiple keywords or keypresses. This class is typically used with menus and forms.
  - **com.audium.core.vfc.util.IfCondition** – This class is used to specify an expression to put inside an if statement. It handles standard numerical and string operations and can support expressions contains *ands* (&&) and *ors* (||).
  - **com.audium.core.vfc.form.UsedInFilled** – This class is a Java interface that is used to identify all the VFCs that can be used inside a `<filled>` tag. It is used simply as a marker for those VFCs.