Orca Technical Reference

Orca Technical Reference

Table of Contents

List of Figures

Foreword

Orca is a flexible, extensible, and powerful assistive technology that provides end-user access to applications and toolkits that support the AT-SPI (e.g., the GNOME desktop). With early input from and continued engagement with its end users, Orca has been designed and implemented by the Sun Microsystems, Inc., Accessibility Program Office.

NOTE: Orca is currently a work in progress. As a result, this and other books in the Orca Documentation Series are under continuous modification and are also in various states of completeness.

This book covers the overall architecture of Orca, including a portion on writing custom scripts. The bulk of the user information and user experience design can be found on the Orca WIKI at http://live.gnome.org/Orca.

Chapter 1. Prerequisites

To help narrow the scope of the Orca development activity, Orca uses existing software where available. For example, as mentioned in the requirements, Orca is a screen reader that needs to be able to interact with speech synthesis, braille, and screen magnification services, but it need not be the provider of such services. Given this, Orca has the following dependencies:

1.1. GNOME 2.22 or better

The GNOME 2.22 desktop contains a number of bug fixes and enhancements to the accessibility infrastructure. These are needed for Orca to run properly. GNOME 2.22 also packages a variety of other dependencies of Orca, including gnome-speech, gnome-mag, pyatspi, etc.

1.2. Python v2.4 or better

Orca is written in the Python programming language and depends upon features found in Python versions 2.4 and greater.

1.3. BrlTTY v3.7.2 or better

BrlTTY [*BRLTTY*>] provides access to a variety of Braille displays, and consists of a library and a daemon to provide programmatic interaction with the display.

1.4. Keyboard Navigation

As much as possible, Orca relies upon the keyboard navigation methods built in to the native platform. For example, it is expected that the native platform will provide access via traditional methods such as the "tab" key, keyboard mnemonics, and keyboard accelerators.

Chapter 2. Architecture

The Orca architecture has been driven primarily by the Orca User Experience Design. There are two primary operating modes of Orca: a focus tracking mode and a flat review mode.

The focus tracking mode generally relies upon applications to provide reasonable keyboard navigation techniques to allow the user to operate the application without requiring the mouse. As the user uses traditional keyboard navigation techniques to move from component to component in the application (e.g., pressing the Tab key to move from pushbutton to text area to toggle button, etc.), Orca will present this to the user via braille, speech, magnification, or a combination thereof. In the cases where more complex navigation is needed, such as structural navigation of complex text documents, Orca also provides a facility to define keyboard and braille input events that it can intercept and handle appropriately.

The flat review mode provides the user with the ability to spatially navigate a window, giving them the ability to explore as well as discover and interact with components in the window. Orca provides a default set of keybindings for flat review, and these keybindings can be easily redefined by the user.

The various modes of Orca are handled by "scripts," which are Python modules that can provide a custom interpretation of an application's interaction model. It is not intended that there will be a unique script for every application. Instead, it is expected that there will be a general purpose script that covers a large number of applications. In the event that more compelling or custom behavior is desired for an application, however, one can use a custom script for the application. Furthermore, scripts can subclass other scripts, allowing them to be quite simple.

As illustrated in the high level Orca architecture diagram, the main components of Orca are as follows: desktop applications that support the AT-SPI, the AT-SPI registry and infrastructure, Orca itself, Orca Scripts, and system services (e.g., speech, braille, magnification).

Figure 2-1. High Level Orca Architecture. The main components of Orca are as follows: desktop applications that support the AT-SPI, the AT-SPI registry and infrastructure, Orca itself, Orca Scripts, and system services. The key communication between the components is depicted.

High Level Orca Architecture

The following sections describe the architecture in more detail.

2.1. Desktop and AT-SPI

Orca's sole view of any application on the desktop is via the AT-SPI [*AT-SPI*>]. The AT-SPI is an IDL/CORBA/Bonobo-based technology [*Bonobo*>] that provides a common interface for the desktop and its applications to expose their GUI component hierarchy to assistive technologies such as Orca.

AT-SPI support is provided by toolkits such as GNOME's GTK+ toolkit (via gail [*GAIL*>]), the Java platform (via the Java access bridge), and the custom toolkits used by applications such as Mozilla and Open Office. Future support includes the Qt toolkit of KDE.

Assistive Technologies interact with the AT-SPI via two primary means: the AT-SPI registry and accessible objects. The AT-SPI registry permits assistive technologies to discover existing applications on the desktop and to register for event notification for AT-SPI events (e.g., window creation, focus changes, object state changes, etc.) and device events (e.g., keyboard input events). Accessible objects provide the assistive technology with information about the application, and tend to mirror the actual GUI component hierarchy. Accessible objects can be obtained in three ways:

1. From the AT-SPI registry via queries on the desktop
2. From an AT-SPI event
3. From another Accessible via parent/child relationships and other relationships such as "label for" and "labeled by".

Orca's interaction with the AT-SPI is managed through Orca's `atspi.py` module. The `atspi.py` module communicates directly with the AT-SPI via the AT-SPI IDL interfaces and also provides a number of classes that help with AT-SPI interaction: `Registry`, `Accessible`, and `Event`. The full documentation for each of these classes is available in the pydoc for Orca while the following paragraphs provide a brief overview.

The `Registry` class provides a singleton class instance to access to the AT-SPI registry. It provides convenience methods for registering AT-SPI event listeners and device event listeners, and also provides the mechanism for starting and stopping event delivery from the AT-SPI registry.

The `Accessible` class provides a wrapper for communicating with CORBA objects that implement the AT-SPI Accessible and Application interfaces. Using Python's ability to add new properties to a class instance at run time, Orca can also annotate Accessible class instances with additional information. The main purpose of an Accessible is to provide a local cache for accessible objects, preventing the need for numerous round trip calls to the AT-SPI registry and application for information.

The `Event` class provides a wrapper for converting AT-SPI events into Python `Event` instances. The main purpose is to convert the AT-SPI accessible source of the event into a Python `Accessible` instance and to also provide an `Event` instance that can be annotated by scripts (the actual AT-SPI event delivered by the registry is read-only).

As illustrated in the high level Orca architecture diagram, the `atspi` module has been used to register event and device listeners with the AT-SPI registry. Each exemplary desktop application (Firefox, NetBeans, GAIM, StarOffice) emits AT-SPI events to the AT-SPI registry which then delivers them to the `atspi` module. The `atspi` module then calls all appropriate listeners for the events it receives from the AT-SPI registry.

In this case, the `orca` module receives keyboard events, which it interprets and also sends on to the `focus_tracking_presenter` module. Of more interest, however, is that the `focus_tracking_presenter` module receives AT-SPI events which it then passes on the script for the application associated with the event. If there is no script, the `focus_tracking_presenter` will create an instance of the default script. See the Orca Script Writing Guide for more information.

The `atspi` module also registers its own set of event listeners that it uses to keep its local cache of accessible objects up to date.

IMPLEMENTATION DETAIL: Because processing AT-SPI object events can be time consuming, and because the notification of AT-SPI object events is relatively "bursty," the `focus_tracking_presenter` maintains a queue of AT-SPI object and input device events. It adds the events to this queue when it receives them and processes the events on the GLib idle handling thread. This permits Orca to survive a relatively long burst of events and also allows it to handle the events on a thread that is compatible with GLib.

2.2. Orca Module

The `orca` module is the "main entry point" of Orca. It initializes the components that Orca uses (atspi, speech, braille, mag) and loads the user's settings. It also is the first to receive all keyboard and braille input events and delves them out to other Orca components appropriately.

The `orca` module maintains the current known "locus of focus" in the `locusOfFocus` field of the `orca_state` module. The `locusOfFocus` is intended to represent the current object that the user is working with. In simple terms, it is the object that is highlighted or has the dotted line drawn around it. Be advised that the notion of "focus" differs from toolkit to toolkit. For example, the object with toolkit focus may actually be the parent of the object that is highlighted. The `locusOfFocus` is an an attempt to neutralize these differences across toolkits: the locus of focus is the individual object that is highlighted, has the caret, etc.

Orca scripts set the locus of focus to inform Orca when the locus of focus has changed. In addition, in the event that there was a visual appearance change to the object that has the locus of focus, the `orca` module provides a `visualAppearanceChanged` that scripts can use to inform Orca of this event.

NOTE: The `orca_state.locusOfFocus` field is intended to be set only via the `setLocusOfFocus` method of the `orca` module. Because the `setLocusOfFocus` method performs bookkeeping and other tasks, the `orca_state.locusOfFocus` field should never be set directly.

2.2.1. settings

The **settings** module (not depicted in the high level Orca architecture diagram) holds preferences set by

the user during configuration. These settings include the following: use of speech and/or braille, voice styles, key echo, text echo, and command echo.

Any Orca module can check the value of a setting by examining the field directly in the `settings` module. In addition, the `orca` module will import the `user-settings` module from the `~/.orca directory`, if it exists (it is created as part of the configuration process that is run the first time Orca is used or when the user presses Insert+Space to invoke the configuration GUI).

The `user-settings` module is a Python script, allowing it to contain functions, class definitions, etc. The primary job of the `user-settings` is to directly set the values of fields in the `settings` module.

IMPLEMENTATION DETAIL: the `init` method of the `orca` module obtains settings. As a result, the `user-settings` module is imported very early in the Orca life cycle.

2.3. Orca Scripts

Internally, the `orca` module keeps track of list of `PresentationManager` instances (see the pydoc for `presentation_manager.py`). The `FocusTrackingPresenter` (see `focus_tracking_presenter`) is of the most interest, as it is the `PresentationManager` that manages scripts.

Details on the `FocusTrackingPresenter` and Orca scripts can be found in the Orca Script Writing Guide.

2.4. System Services

Orca relies on existing system services to provide support for speech synthesis, braille, and screen magnification. To interact with these services, Orca provides the modules described in the following sections.

2.4.1. speech

The **speech** module provides Orca's Python interface to system speech services. Each speech service is generated by a "speech server factory." There are currently two such factories: one for [*gnome-speech*>] (see `gnomespeechfactory.py` and one for [*Emacspeak*>] (see `espeechfactory.py`), though it is expected that support for other factories can be added in the future.

Each speech factory offers up a list of `SpeechServer`s, where each `SpeechServer` is typically an interface to a particular speech engine. For example, the `espeechfactory` will offer up a `SpeechServer` that talks to the Fonix DECtalk engine and a `SpeechServer` that talks to the IBMTTS engine. Likewise, the `gnomespeechfactory` will offer up a `SpeechServer` that uses the `gnome-speech` interface to talk to the Festival Speech Synthesis System, a separate `SpeechServer` that also uses the `gnome-speech` interface to talk to the Fonix DECtalk engine, and so on.

Each `SpeechServer` instance then provides a set of methods for actually speaking. Each of the methods accepts an `ACSS` instance, which represents an aural cascading style sheet ([*ACSS*>]) that defines the voice and voice parameter settings to use.

As part of the `orca-setup` process, the user selects a particular speech factory, `SpeechServer`, and voice to use as their default voice. When Orca starts, the `speech` module looks for these settings and connects to the appropriate speech factory and `SpeechServer`. In the event the a connection cannot be made, the `speech` module attempts to find a working synthesis engine to use by examining its list of speech factories. The `speech` module then provides simple methods that delegate to the `SpeechServer` instance. This model allows scripts to use their own `SpeechServer` instances if they wish, but scripts typically just rely upon the user's default preferences.

2.4.2. braille

The **braille** module provides Orca's Python interface to the system's BrlTTY [*BRLTTY*>] daemon. The BrlTTY daemon, in turn, provides the interface to braille devices for both displaying braille and receiving input from the user.

TODO: flesh this section out more.

2.4.3. mag

The **mag** module provides Orca's Python interface to the system's gnome-mag [*gnome-mag*>] CORBA service(s). The magnification component provides methods that permit Orca discover screen magnification services and set their desktop region of interest.

TODO: flesh this section out more.

Chapter 3. Introduction to Scripting

In this section, you will learn more about the Orca architecture as well as how to create your own custom scripts for Orca.

The goal of scripting is to provide Orca with the capability of providing a natural feeling and compelling user experience for the various user interaction models of different desktop applications.

The Orca scripting approach allows scripts to extend and/or override the behavior of other scripts, thus simplifying the job of a script writer. To further facilitate script writing, Orca provides a "default" script that provides a reasonable default behavior for Orca. This will not only serve as the "fallback script" for Orca, but will also typically serve as the "jumping off" point for writing custom scripts. Furthermore, keep in mind that the "default" script is intended to cover a large variety of applications. As such, you may find that it is not necessary to write a custom script.

The primary operating mode of Orca is "focus tracking mode," where Orca keeps track of the most relevant user interface object that has keyboard focus. When Orca detects changes to this object, which Orca refers to as the "locus of focus," Orca will present relevant information to the user.

As such, the primary goal of a script is to assist Orca in tracking of the locus of focus as well as presenting information about the locus of focus. A script does this by registering for one or more AT-SPI events and then reacting appropriately when it receives those events. A script can also intercept and interpret keystrokes and braille input events, allowing it to further extend the behavior of Orca.

3.1. Script Contract

The contract for a script is documented in detail in the pydoc of the `Script` class in the `script.py` module. The `Script` subclass defined in the `default.py` module provides the default behavior for Orca when it encounters applications and toolkits that behave like the GTK toolkit. It is expected that new scripts will typically extend the `Script` subclass of the `default.py` module rather than directly extending the `Script` class defined in the `script.py` module.

3.2. Script Life Cycle

BIRTH: Orca's `focus_tracking_presenter` module is the sole maintainer of scripts. Whenever it receives an event from the AT-SPI Registry, the `focus_tracking_presenter` will determine the application associated with that event and create a new script for that application if on has not yet been created. Only one script instance per application instance is allowed by the `focus_tracking_presenter`.

The script creation process consists of the following steps:

- The `focus_tracking_presenter` will attempt to perform a Python `import` using the application name as the name of an Orca module. For example, for the `gnome-terminal` application, the `focus_tracking_presenter` will look for the `gnome-terminal.py` in the `orca.scripts` package (see the script naming discussion in the debug utilities section to determine what to name your script). If it cannot find such a module in the Python search path, the `focus_tracking_presenter` will then check in the `orca` package for a module matching the name of the toolkit used by the application. Failing that, Orca will create an instance of the `Script` class defined in the `default.py` module.

  NOTE: the `focus_tracking_presenter` also maintains a table to map application names to script names. This is useful in many cases, such as if the application name changes over time or the application contains characters that are awkward in file system names. To extend or override this table, one can call the `setScriptMapping` method of the `settings` module.

  IMPLEMENTATION DETAIL: it is possible to tell Orca to bypass all custom script creation by setting `orca.settings.enableCustomScripts=False` in your `~/.orca/user-settings.py` module. This can be useful for debugging purposes.

- Each script module is expected to provide a `Script` class that ultimately extends the `orca.Script` class defined in the `script.py` module. The constructor takes the accessible application object as an argument.

  The constructor for the `Script` instance is expected to define any keystrokes, braille buttons, and AT-SPI event listeners it is interested in (see the Customized Behavior section for how to do this).

- Once it has created a script, the `focus_tracking_presenter` will register event listeners for all AT-SPI events associated with script (i.e., the script should not register these events itself). When the `focus_tracking_presenter` receives the events, it will pass the event to the script associated with the event, regardless if the application associated with the script has focus or not.

  IMPLEMENTATION DETAIL: the `focus_tracking_presenter` registers its own `processObjectEvent` method as the AT-SPI event listener. This method finds (and creates if necessary) the script associated with the event and passes the event onto the required `processObjectEvent` method of the script for processing. Each `Event` (see the `atspi` module) has the following fields:

  - `source`: an `Accessible` (see the `atspi` module) instance representing the object associated with the event
  - `type`: a string describing the event (e.g., `window:activated`)
  - `detail1` and `detail2`: integer details for the event (see the AT-SPI documentation)
  - `any_data`: something associated with the event (see the AT-SPI documentation)

- The `focus_tracking_presenter` also keeps track of the active script (as determined by the script associated with the currently active window) and will pass all keyboard and braille input events to the active script.

  IMPLEMENTATION DETAIL: the `focus_tracking_presenter` implements the `processKeyboardEvent` and `processBrailleEvent` methods which are called by the main `orca` module whenever it receives a keystroke or braille input event. The `focus_tracking_presenter` will pass these events onto the `processKeyboardEvent` and `processBrailleEvent` methods of the active script.

  IMPLEMENTATION DETAIL: Because processing AT-SPI object events can be time consuming, and because the notification of AT-SPI object events is relatively "bursty," the `focus_tracking_presenter` maintains a queue of AT-SPI object and input device events. It adds the events to this queue when it receives them and processes the events on the GLib idle handling thread. This permits Orca to survive a relatively long burst of events and also allows it to handle the events on a thread that is compatible with GLib.

LIFE: Whenever a script receives an event, the script can do whatever it wants. Its primary task, however, is to assist Orca in keeping track of the locus of focus. When a script detects a change in the locus of focus, it should call `orca.setLocusOfFocus` with the `Accessible` object instance that is the new locus of focus. Among other things, this results in the `orca_state.locusOfFocus` field being updated.

NOTE: The `orca_state.locusOfFocus` field is intended to be set only via the `setLocusOfFocus` method of the `orca` module. Because the `setLocusOfFocus` method performs bookkeeping and other tasks, the `orca_state.locusOfFocus` field should never be set directly.

IMPLEMENTATION DETAIL: The `orca` module has logic to detect if the locus of focus really changed and will propagate the change on as appropriate. The `orca.setLocusOfFocus` method first sends the change to the `locusOfFocusChanged` method of the `focus_tracking_presenter`, which then passes the change onto the required `locusOfFocusChanged` method of the active script. The `locusOfFocusChanged` method is the primary place where a script will present information to the user.

In many cases, the locus of focus doesn't change, but some property of the current locus of focus changes. For example, a checkbox is checked or unchecked, yet remains as the locus of focus. In these cases, a script should also keep Orca informed by calling `orca.visualAppearanceChanged`.

IMPLEMENTATION DETAIL: Like the `locusOfFocusChanged` method, the `visualAppearanceChanged` method of the `orca` module will first call the `visualAppearanceChanged` method of the `focus_tracking_presenter`, which will then call the required `visualAppearanceChanged` of the active script. The `visualAppearanceChanged` is the primary place where a script will present such information to the user.

DEATH: Whenever the `focus_tracking_presenter` detects that an application has gone away (by determining that the application has been removed from the desktop), it will delete the script for that application and unregister any event listeners associated with that script.

IMPLEMENTATION DETAIL: the `focus_tracking_presenter` determines an application has gone away by detecting a `object:children-changed:remove` event on the desktop.

Chapter 4. Customized Behavior

NOTE: THIS WILL CHANGE POST V1.0. In particular, the method for setting up event handlers and keyboard/braille bindings will be changed so as to allow for easier customization of these bindings. As such, the information in this chapter is here only for historical purposes.

The customized behavior of a script is set up in its constructor. In its constructor, each script is expected to extend/override several fields as illustrated in the following diagram and describe below:

Figure 4-1. Orca Script Diagram

Orca Script Diagram

- `listeners`: a dictionary where the keys are strings that match AT-SPI event types (e.g., `focus:`, `object:text-caret-moved`, etc.), and the values are functions to handle the event. Each function is passed an `Event` instance (see the `atspi.py` module) as its sole parameter and no return value is expected.
- `keybindings`: an instance of `keybindings.KeyBindings` (see the `keybindings.py` module) that defines the keystrokes the script is interested in.
- `braillebindings`: a dictionary where the keys are BrlTTY commands (e.g., `CMD_HWINLT`, defined in `braille.py`), and the values are `InputEventHandler` instances.

The constructor for the `Script` class, which all scripts should ultimately extend (most will extend the `Script` subclass of the `default.py` module, which in turn extends `Script` class of the `script.py` module), sets up empty values for each of these fields. As such, a subclass merely needs to extend/override these fields. Each of these fields is described in more detail in the following sections.

4.1. Defining Event Listeners

As described above, the `listeners` field is a dictionary where the keys are strings that match AT-SPI event types (e.g., `focus:`, `object:text-caret-moved`, etc.), and the values are functions to handle the event. A script's constructor can modify/extend this dictionary by merely defining an entry:

```
self.listeners["focus:"] = self.onFocus
```

In the event there is already an entry in the `listeners` dictionary, it will be overridden by the new value.

As described previously, the `focus_tracking_presenter` will register listeners on behalf of a script, and will notify the script of any events via the script's `processObjectEvent` method. The `processObjectEvent` method of the top level `Script` class examines the `type` field of the given

event, calling any matching functions from the listeners dictionary. As such, it is unlikely that a Script subclass will ever need to override the processObjectEvent method. Instead, it merely needs to populate the listeners dictionary as appropriate.

The function for an event listener merely takes an Event instance (see the atspi.py module) and does whatever it wants; the return value is ignored. For example, the function definition associated with the above listeners entry might look like the following, where the event is described above:

```
def onFocus(self, event):
    """Called whenever an object gets focus.

    Arguments:
    - event: the Event
    """

    ...
    orca.setLocusOfFocus(event, event.source)
    ...
```

4.2. Input Event Handlers

Before describing how to set up keyboard and braille event handlers, it is import to first understand the InputEventHandler, which is defined in the input_event.py module. InputEventHandlers serve a purpose of holding a function to call for a particular input event, and a human consumable string that provides a short description of the function's behavior. The main purpose of the InputEventHandler is to provide support for the "learn mode" of Orca. If learn mode is enabled, the input event handler will consume the input event (i.e., return True) and merely speak and braille the human consumable string. If learn mode is not enabled, the input event handler will pass the active script and the input event on to the function, returning the boolean value of the function as indication of whether the event should be consumed by Orca or passed on to the application.

The best place to look for examples of InputEventHandlers is in the default.py module. For example, this module defines an input event handler for telling the flat review context to move to the home position of a window:

```
reviewHomeHandler = input_event.InputEventHandler(
            Script.reviewHome,
            _("Moves flat review to the home position."))
```

In this definition, default.py is creating an InputEventHandler instance whose function is the Script's method, reviewHome and whose human consumable text describes what will happen. The Script's reviewHome method is defined as follows:

```
def reviewHome(self, inputEvent):
    """Moves the flat review context to the top left of the current
    window."""
```

```
    context = self.getFlatReviewContext()
    context.goBegin()
    self.reviewCurrentLine(inputEvent)
    self.targetCursorCell = braille.cursorCell
    return True
```

Note that the method returns `True` to indicate the input event has been consumed.

4.3. Defining Keyboard Bindings

The keyboard bindings for a script are held in the `keybindings` field, which is a `KeyBindings` instance. This field maintains a set of `KeyBinding` instances.

Keyboard bindings merely define the keystroke and modifier circumstances needed to invoke an `InputEventHandler` instance. This definition is held in a `KeyBinding` instance (see the `keybindings.py` module):

```
self.keybindings.add(
    keybindings.KeyBinding("KP_7",
                           1 << orca.MODIFIER_ORCA,
                           1 << orca.MODIFIER_ORCA,
                           reviewHomeHandler))
```

The first parameter of a `KeyBinding` is a string that represents an X Window System KeySym string for the key. This is typically a string from `/usr/include/X11/keysymdef.h` with the preceding 'XK_' removed (e.g., XK_KP_Enter becomes the string 'KP_Enter'), and is used as a means to express the physical key associated with the KeySym.

The second parameter is a bit mask that defines which modifiers the keybinding cares about. If it does not care about any modifier state, then this mask can be set to 0. In the example above, the keybinding is being told to pay attention to the `MODIFIER_ORCA` modifier, which is a modifier Orca sets when the "Insert" key is pressed. Other examples of modifier bit positions include those defined in the AT-SPI Accessibility specification: MODIFIER_SHIFT, MODIFIER_SHIFTLOCK, MODIFIER_CONTROL, MODIFIER_ALT, MODIFIER_META, MODIFIER_META2, MODIFIER_META3, and MODIFIER_NUMLOCK. These can be obtained via the `orca.atspi.Accessibility` field. For example, `orca.atspi.Accessibility.MODIFIER_SHIFTLOCK`.

The third parameter is a bit mask that defines what the modifier settings must be. If a bit is set, it means the associated modifier must be set. The only meaningful bits in this mask are those that are defined by the second parameter. In the example above, the keybinding cares about the `MODIFIER_ORCA` modifier, and the third parameter says this modifier must be set.

The last parameter is the `InputEventHandler` to us if the user types a keystroke qualified by the previous parameters. `InputEventHandler`s are described in the previous section.

4.4. Defining Braille Bindings

Refreshable braille displays have buttons that the user can press. The BrlTTY system provides a means for standardizing the types of input events one can generate using these buttons, and a script can set up braille bindings to handle these events.

The braille bindings for a script are held in the `braillebindings` field, which is a dictionary. The keys for the dictionary are BrlTTY constants representing braille input events (see `braille.py` for a list), and the values are `InputEventHandler` instances:

```
self.braillebindings[braille.CMD_TOP_LEFT] = reviewHomeHandler
```

In the above example, the BrlTTY `braille.CMD_TOP_LEFT` input event has been set to be handled by the same `reviewHomeHandler` instance described previously.

Chapter 5. Script Utilities

There are many common things a script wants to do: generate speech, update braille, etc. In addition, there are many common things a script writer wants to do, especially getting debug output to determine just what the AT-SPI is sending it. This chapter discusses the debug utilities of Orca as well as a variety of utilities to assist a script in managing speech, braille, and magnification.

5.1. Debug Utilities

The debug utilities (defined in the `debug.py` module) of Orca provide a means for selectively turning on information to be sent to the console where Orca is running. This information is quite useful in determining what is happening inside Orca as well as what the AT-SPI is sending to Orca.

Let's begin the discussion of the debug utilities with the top question on any script writer's mind: "What do I name my script?" As you may recall, the name of a script is based upon the name of the application as given to us by the AT-SPI. One of the easy ways to determine this is to listen for `window:activate` events that will be issued when an application is started. These events can then be used to determine the name of the application.

Fortunately, the `focus_tracking_presenter` already registers for `window:activate` events, so all you need to do is tell Orca to print these events out when it receives them. The method for doing this involves telling the debug utilities what to do, and this can be done by modifying your `~/.orca/user-settings.py`.

There are two main settings to tell Orca to print out events: an event filter and an event debug level. The event filter is a regular expression that is used to match AT-SPI event types, and the event debug level specifies a threshold for when to actually print information to the console (for more complete detail on these settings, refer to `debug.py`). These settings can be modified by adding the following lines to your `~/.orca/user-settings.py`:

```
orca.debug.setEventDebugFilter(re.compile('window:activate'))
orca.debug.setEventDebugLevel(debug.LEVEL_OFF)
```

Now, when you rerun Orca, it will output information whenever it receives a `window:activate` event from the AT-SPI registry. For example, if you run Star Office, you should see output similar to the following:

```
OBJECT EVENT: window:activate detail=(0,0)
              app='StarOffice' name='StarOffice' role='frame'
              state='ENABLED FOCUSABLE RESIZABLE SENSITIVE SHOWING VISIBLE'
```

The string `app='StarOffice'` indicates the name of the application is 'StarOffice.' As such, if you wanted to write a custom script, you would call it `StarOffice.py`.

NOTE: you can also get other information while Orca is running by pressing the debug keystrokes:

- Insert+F5: dump a list of all applications to the console
- Insert+F6: speak/braille information about the active script and application with focus
- Insert+F7: dump the ancestors of the object with focus to the console
- Insert+F8: dump the entire widget hierarchy of the application with focus to the console

The debug module also includes a number of other methods, each of which is described in more detail in `debug.py`. Note that each method includes a debug level threshold. The `debug.py` module has a description of various level settings and what to expect for output.

- `setDebugLevel(newLevel)`: sets the debug level threshold, turning on or off the various debug code built in to the various Orca modules. This is typically called from `~/.orca/user-settings.py`.
- `setEventDebugLevel(newLevel)`: described above; typically called from `~/.orca/user-settings.py`.
- `setEventDebugFilter(regExpression)`: described above; typically called from `~/.orca/user-settings.py`.
- `printException(level)`: if an exception is caught, this can be used to print out detail about it
- `printStack(level)`: prints the current stack; useful for determining when and why a code path is being executed
- `println(level, text)`: prints the given text; useful for general debug output
- `printObjectEvent(level, event)`: prints out the given AT-SPI event
- `printObjectEvent(level, event)`: prints out the given AT-SPI event, using the event debug level as an additional threshold; this is already used by the `focus_tracking_presenter`, so you are unlikely to need it
- `printInputEvent(level, string)`: prints out the given AT-SPI event, using the event debug level as an additional threshold; this is already used by `orca.py` (for keyboard events) and `braille.py` (for braille events), so you are unlikely to need it

NOTE: One debug level of interest is `debug.LEVEL_FINE`. This level will tell you when a script is activated, and can be useful to determine if Orca is actually finding your script! For example, when the script for the `gnome-terminal` is activated by the `focus_tracking_presenter`, you will see the following output:

```
ACTIVE SCRIPT: gnome-terminal (module=orca.scripts.gnome-terminal)
```

Notice that the class of the script instance is included. If you determine that this class is not what you expect when you are developing your custom script, then something went wrong when trying to find or

load your custom script. This can often happen because Python performs a lot of late binding and compilation, thus errors are often not encountered until a specific code path is executed at run time. You can tell the `focus_tracking_presenter` to give you more information about any possible failures or exceptions it handles in this area by setting the debug level to `debug.LEVEL_FINEST`.

5.2. Speech Synthesis

Orca provides two main modules for speech output: `speech.py` and `speechgenerator.py`. The `speech.py` module provides the main interface to the speech synthesis subsystem. The `speechgenerator.py` module provides a `SpeechGenerator` class that can be used to to actually generate the text to be spoken for various objects. The expected use of the two modules is as follows: a script will create its own instance of a `SpeechGenerator` and will use it to generate text. The script will then pass this text to the `speech.py` module to be spoken.

5.2.1. `speech.py`

For the purposes of script writing, the main entry points of the `speech.py` module are `speak`, `speakUtterances`, and `stop`

See the `speech.py` module for more information.

5.2.2. `speechgenerator.py`

The primary goal of a `SpeechGenerator` is to create text to be spoken for an accessible object. There are two public entry points into a `SpeechGenerator`:

- `getSpeech(obj, already_focused)`: returns a list of strings to be spoken for the given accessible object. The `already_focused` boolean parameter provides a hint to the speech generator about how much text to generate. For example, if a check box that already has focus is to be spoken, usually the reason for this is that the state changed between checked and unchecked. As a result, an appropriate thing to do in this situation is to only speak the new change in state (e.g., "checked").
- `getSpeechContext(obj, stopAncestor)`: returns a list of strings to be spoken that describe the visual context of the given accessible object. This is loosely represented by the hierarchical relationship of the object (i.e., the "Quit" button in the "File" menu in the ...), and the amount of information can be contained by specifying an accessible `stopAncestor` above which we do not want to know anything about. The primary use of this method is to provide the user with feedback regarding the relevant visual context information that changed when the locus of focus changes, but this method is also useful for assisting in "where am I" queries.

NOTE: Orca currently provides some level of support for verbosity via the `VERBOSITY_LEVEL` fields of the `settings.py` module. There are currently two verbosity levels: `VERBOSITY_LEVEL_BRIEF` and

VERBOSITY_LEVEL_VERBOSE. A SpeechGenerator subclass is expected to examine the speechVerbosityLevel property of the settings.py module and provide the appropriate level of text:

```
if settings.speechVerbosityLevel == settings.VERBOSITY_LEVEL_VERBOSE:
    utterances.append(rolenames.getSpeechForRoleName(obj))
```

5.3. Braille Output

Like speech, Orca provides two main modules for braille: braille.py and braillegenerator.py. The braille.py module provides the main interface to the braille display. The braillegenerator.py module provides a BrailleGenerator class that can be used to to actually generate the text to be displayed for various objects. The expected use of the two modules is as follows: a script will create its own instance of a BrailleGenerator and will use it to braille regions. The script will then pass these braille regions to the braille.py module to be displayed.

5.3.1. braille.py

TODO: [[[WDW - much writing to be done here, especially regarding how regions will provide automatic support for cursor routing keys.]]]

5.3.2. braillegenerator.py

The primary goal of a BrailleGenerator is to create text to be displayed for an accessible object. There are two public entry points into a BrailleGenerator:

- getBrailleRegions(obj, groupChildren=True): returns a list of two items: the first is an ordered list of braille Region instances that represent text to be displayed on the braille display, left-to-right on one line; and the second is an element from the first list that represents which Region has "focus" and should be represented by the braille cursor on the display.

  TODO: [[[WDW - describe grouping of children.]]]

- getBrailleContext(obj): returns an ordered list (i.e., an array) of braille Region instances that describe the visual context of the given accessible object. This is loosely represented by the hierarchical relationship of the object (i.e., the "Quit" button in the "File" menu in the ...).

Typically, a script will "build up" a single logical line of text for the braille display. The beginning of this line will be the result of the call to getBrailleContext and the remainder of the line will be the result

of one or more calls to `getBrailleRegions`. Since the logical line will typically be longer than the number of cells on the braille display, the `braille.py` module will scroll to show the braille `Region` with focus. Furthermore, the `braille.py` will also respond to BrlTTY input events to allow the user to use braille display input buttons for scrolling to review the entire line.

NOTE: Orca currently provides some level of support for verbosity via the `VERBOSITY_LEVEL` fields of the `settings.py` module. There are currently two verbosity levels: `VERBOSITY_LEVEL_BRIEF` and `VERBOSITY_LEVEL_VERBOSE`. A `BrailleGenerator` subclass is expected to examine the `brailleVerbosityLevel` property of the `settings.py` module and provide the appropriate level of text:

```
if settings.brailleVerbosityLevel == settings.VERBOSITY_LEVEL_VERBOSE:
    regions.append(braille.Region(
                    " " + rolenames.getBrailleForRoleName(obj)))
```

Chapter 6. Internationalization (I18N) Support

All human-consumable text obtained from AT-SPI calls is expected to be in a localized form. As such, Orca does not do any extra localization processing when working with text obtained via the AT-SPI.

For text generated by Orca itself, Orca handles internationalization and localization using the [*gettext*>] support of Python. The gettext support of Python is similar to the GNU gettext module. Each human consumable string of Orca is US English text wrapped in a call to gettext.gettext. The call to gettext.gettext will either return a localized string or default to the US English text. Orca depends upon an active and thriving community of open source translators to provide the localizations.

The synthesis of localized speech is to be provided by the underlying gnome-speech engine. That is, Orca merely passes localized text to the speech engine, which is responsible for the correct interpretation and pronunciation.

The generation of localized braille is to be determined. *TODO:* BrlTTY currently does not support this at the moment, but it is expected that the BrlTTY developers will add this in the future.

Bibliography

[AT-SPI] Bill Haneman, Marc Mulcahy, and Michael Meeks, *AT-SPI (http://directory.fsf.org/accessibility/at-spi.html)* .

[ACSS] T.V. Raman, *Aural Style Sheets (http://www.w3.org/TR/1998/REC-CSS2-19980512/aural.html)* .

[Bonobo] George Lebl, *Gnomenclature: Intro to bonobo (http://lidn.sourceforge.net/articles/gnomenclatureintrotobonobo/)* .

[BRLTTY] Dave Meilke, Nicolas Pitre, and Stephane Doyon, *BRLTTY (http://directory.fsf.org/accessibility/brltty.html)* .

[Emacspeak] T.V. Raman, *Emacspeak (http://emacspeak.sourceforge.net/)* .

[GAIL] Bill Haneman, *GAIL (http://freshmeat.net/projects/gail/)* .

[gettext] TODO: Unknown, *gettext (http://docs.python.org/lib/module-gettext.html)* .

[gnome-mag] Bill Haneman, *gnome-mag (http://directory.fsf.org/accessibility/gnome-mag.html)* .

[gnome-speech] Marc Mulcahy and Michael Meeks, *gnome-speech (http://directory.fsf.org/accessibility/gnome-speech.html)* .

[Gnopernicus] Remus Draica, *Gnopernicus (http://directory.fsf.org/accessibility/gnopernicus.html)* .

[JAWS] Freedom Scientific, *JAWS (http://www.freedomscientific.com/fsproducts/softwarejaws.asp)* .

[XKB] Erik Fortune, William Walker, Donna Converse, and George Sachs, *The XKB keyboard extension (http://matrix.netsoc.tcd.ie/hcksplat/work/XKBlib.pdf)* .