Developer's guide

SigTest Tool Developer's Guide

This document helps a new developer understand the SigTest tool source code.

Table of Contents

Basic Conventions

Source files are text files in ISO8859-1 format, always in English. Tab characters are defined as eight spaces. The standard indentation unit is four spaces. It is common to configure your editor to insert four spaces for each press of the Tab key and substitute a tab character for every two of those key presses. *Only use space characters for indentation.*

Eighty character lines are not required. It is best to keep lines under 80 columns if possible, although longer lines are acceptable if it makes the code more readable. It is common to keep a long literal string on a single line and go to a new line at the next opportunity, usually for the next parameter. *Do not allow your editor to reformat all the code in a file to eighty columns because of word wrapping.* Make your own decision as to whether lines should be wrapped for display purposes.

Method definitions and calls do not have a space between the method name and the parentheses that surround the argument list. For conditional statements there is a space after the keyword and before the condition list (`while`, `for`, `if`, ...) Curley brackets begin at the end of the line that defines the context and end on their own line, aligned to the left justification of the beginning context line. Comment the ending brackets when there is a possibility for confusion. This comment goes at the next four character indent for that line (see the example below).

Example:

```
public void exampleMethod(boolean foo) {
    if (foo) {
                while (x) {
                    x = method1(foo, bar);
                }        // while
    }
}   // exampleMethod()
```

Internationalization

All code in the core of the SigTest tool and utilities that may be distributed externally must be internationalized. For the majority of development work, this just means that literal strings must be retrieved from a resource bundle. This applies to strings being written for end user debugging, file output (not data files), warning and error messages, and all strings in the GUI.

It is important to follow the established coding patterns maintain consistency in the code and to facilitate automated string checking in the build (discussed below).

To retrieve a literal string from a resource bundle (usually in non-GUI code) use this coding pattern:

```
import com.sun.javatest.util.I18NResourceBundle;

class .... {
    public String exMethod() {
        return i18n.getString("allTestsFilter.name");
    }

    private static I18NResourceBundle i18n = I18NResourceBundle.getBundleForClass(Harness.class);
}
```

The `I18NResourceBundle` class knows how to locate the correct resource bundle at runtime. It is static because the bundle is static once the correct bundle has been located (determined at runtime). Strings are always located in the default locale bundle `i18n.properties`. There is one bundle for each package. The `I18NResourceBundle`class finds the right bundle for you based on the class you specify.

The name space inside the bundle is up to you. For example, the prefix for the `BranchPanel` class is `bp.*`. In the code above, `allTestsFilter` is the prefix for strings used by the `AllTestsFilter`class. See the existing source code in the workspace for more examples. You generally only need to decide this when you create a new class.

Note that the build has an I18N test that verifies that all keys referenced by calls to `i18n.getI18NString()` exist in the `i18n.properties` bundle. It also does the reverse check to see if there are orphan entries in the bundle. These are both fatal build errors, which is why running the `sigtest-qc` build target is important.

Common Errors

The most common error is forgetting to add an entry to the resource file for GUI components. The build will catch this problem and notify you.