SimpleEdit - an API-Introduction

Patrick Gotthardt

February 8, 2006

Abstract

This documents intention is to tell you how to use the SimpleEdit-API, thus it is written for Java-Developers that'd like to work on the core of it (or write some plugins).

You'll learn about the most fundamental APIs that you'll have to know if you'd like to work with SimpleEdit.

Contents

1 org.simpleedit.SimpleEdit - the Application-class

This is the Application-class of SimpleEdit. It also acts as EventBus for all application-level events, but we'll come to this point later when we discuss the Application-Event-System.

Anyway, it might be a very good idea to make yourself familiar with this class. It offers some utility-functions (I18N-related) that could be of great use for you.

As an example: Using the getBundle-Method, you can easily load a ResourceBundle for your specific class:

```
ResourceBundle bundle = SimpleEdit.getBundle(MyClass.class);
```

This will look for a bundle named "Bundle.properties" (or related Bundles - just like the ResourceBundle-mechanism works) within the package of `MyClass`. In case you don't want to get a complete bundle, you can always call `SimpleEdit.getString(Class clazz, String key, String ... args`.

2 org.simpleedit.WindowManager - the entry-point for every window

Every DockableFrame (that's what the dockables are called in our docking-framework) should be a Singleton if it is somehow possible. There are, of course, cases where it'd be impossible to use it as an singleton, but if it is somehow possible, try to apply to this standard. Singletons are used all over the place in SimpleEdit.

This class supports you with the Singleton-concept as it provides the methods to access the instance of an DockableFrame. Do not use anything else for this purpose. `WindowManager` keeps a cache of instances of the initiated windows, but it keeps it locally, within itself, thus it won't know anything about other Singleton-Style accesses you offer or use. Access a DockableFrame that is to be used as a Singleton **only from WindowManager**.

```
// somewhere
public class MyDockableFrame extends DockableFrame {
    // ...
}
```

```
// to access it
MyDockableFrame frame = WindowManager.get(MyDockableFrame.class);
// do something with it - for example: add it to the Application
// (if you've not done it already)
WindowManager.getManager().addFrame(frame);
// you'll most likely also want to add a menuitem for it in the "view"-menu:
WindowManager.addWindowAction(frame);
```

Every DockableFrame must have a unique ID by which it is accessable through the DockingManager returned by `WindowManager.getManager()`. We believe using Strings for this task is somewhat risky and a source of errors, thus it is discouraged to use them. Use the Singleton-Access whenever possible.

3 org.simpleedit.Lookup - the ServiceProvider

This section is most propably the most important. You can't extend SimpleEdit without knowing about the `Lookup`-class. The `Lookup`-class is an implementation of the ServiceProvider-pattern, thus it is used for decoupling of parts of SimpleEdit to allow others to easily extend and modify them. I've been inspired by the NetBeans-Lookup-mechanism while writing the API for this one, but I've taken my own route when it came to implementation details. Let's examine what this means for you.

The Lookup class itself is an abstract class that'll provide a way to get a default implementation of it. Most likely you'll be using DefaultLookup. I'll give you an example:

```
FileContext ctx = Lookup.getDefault().lookup(FileContext.class,
                  someFile);
```

You don't need to know from where we got that `FileContext`-class. All you need to know is that you got one that works in your context (`someFile`). Note that it is possible that you won't get an service you requested. It is possible, though unlikely as it is encouraged to provide an default implementation for every service that works for every context.

While it is true that most of the time you don't need to know where you got that service from, it is important to know how to provide a way to access your service if you want to provide one.

Let's imagine you would want to provide a service called "MyFunnyService" that would offer a method "myFunnyMethod". What would you have to

do to integrate it? The first step is to implement/define the MyFunnyService-class. This might be an interface, but it could also be a concrete class (as FileContext is).

```
public interface MyFunnyService {
    public void myFunnyMethod();
}
```

The next step would be to implement a ServiceProvider for it. This one should return a MyFunnyService regardless of the specified context (actually the context might be `null`) anyway. A ServiceProvider is supposed to return `null` if it isn't responsible within this context.

```
public class MyFunnyServiceProvider implements ServiceProvider,
                        MyFunnyService {
    public Object getService(Object context) {
        return this;
    }

    public void myFunnyMethod() {
        System.out.println("I'm so funny!");
    }
}
```

As you can see this ServiceProvider implements the Service-Interface himself. In most cases you won't want to do this, but in case you want to: It's possible.

Now, how do you register it? That's easy:

```
Lookup.getDefault().register(MyFunnyService.class,
                new MyFunnyServiceProvider());
```

And from that second on, you can use it like this:

```
Lookup.getDefault().lookup(MyFunnyService.class).myFunnyMethod();
```

If you're offering some service you should document it for other developers so they know how to extend/modify your code. This is especially important for the core of SimpleEdit.

It might be worth to point out that every class can have its own Lookup. If it offers a Lookup, it should implement the `org.simpleedit.LookupProvider`-interface.

4 org.simpleedit.Registry - To store your configuration

This class is pretty important as well. It is highly discouraged to use Properties or some other Java-internal API to store your module configuration. Instead you should be using this extremly powerful API.

Internally it uses the XStream-API to serialize every object you'd like to store within it, thus it enables you to store every data-model directly. The usage is pretty simple, too.

```
// get the registry for your plugin / application part
Registry reg = Registry.getInstance("my.plugin");
// get data data
System.out.printf("You've started this plugin %s times!%n",
            reg.getInt("startcounter", 0)+1);
// set data
reg.put("startcounter", reg.getInt("startcounter", 0)+1);
// store it to disk
reg.store();
```

As I said: You could possibly put everything into it. It's worth to read the documentation of this class. Actually it is one of those classes that've already been documented yet. Most noticable: It implements the Map-interface, thus it has an known API.

5 The EventBus

When we've discussed the SimpleEdit-class I already mentioned that it was the Entry-point to the ApplicationEvent-system. I won't talk about why we need such a system, but rather explain what it does and how you can use it.

Certain parts of SimpleEdit send or listen to specific ApplicationEvents. For example: The filebrowser will send an `FileOpeningRequestEvent` once a file is selected to be opened. The EditorPanel (the component that contains the editors) will listen to this event and open an Editor for this request - by using the Lookup-API we discussed earlier.

Ok, how do you use it? I know that I'm repeating myself, but I've to say it once again: It's extemly easy.

```
// define the event class (or reuse an existing one)
public class MyCustomEvent extends ApplicationEvent {
```

```
    // ...
}

// fire it somewhere
SimpleEdit.fireApplicationEvent(new MyCustomEvent());
```

This is, how you can define and fire a custom ApplicationEvent.

Now, how do you listen to it? There are two ways to do this. And one of them has an descendant, thus we're at three ways. There are good reasons for this amount of different ways to get one task done. The preferred way to do it is by using Annotations.

```
// define the listener
public class MyCustomEventHandler extends ApplicationHandler {
    @EventHandler(MyCustomEvent.class)
    public void someMethod(MyCustomEvent event) {
        // do something
    }
}

// register it (automatic inspection of all important events)
SimpleEdit.addApplicationListener(new MyCustomEventHandler());
// register it (manual specification of events
// - the list of events is a var-arg)
SimpleEdit.addApplicationListener(new MyCustomEventHandler(),
            MyCustomEvent.class);
```

Both ways to register a listener in the Bus is fine. Now, this approach introduces problems when you don't want to/can't extend AnnotationEventListener. In this case you'd have to change the registration code to this (for the listener itself: just erase the extends-stuff):

```
SimpleEdit.addApplicationListener(new ApplicationHandler(
                new MyCustomEventHandler()));
```

That's it. Ok... I said there was a third way, and here it is: ApplicationHandler is just an implementation of the ApplicationListener. If you want to listen to multiple events in just one method, this is the way to go: Implement your own ApplicationListener.

The above code would look like this:

```
// define the listener
```

```
public class MyCustomEventHandler implements ApplicationListener {
    public void handleEvent(ApplicationEvent event) {
        // do something
    }
}

// register it (this time this is the only way to do it!)
SimpleEdit.addApplicationListener(new MyCustomEventHandler(),
                    MyCustomEvent.class);
```

As our Annotation-based system works on classes there shouldn't be any risk of runtime-problems, thus it is the preferred way (including automatic-registration). Use the low-level way whenever you can't use one of the Annotation-based ways.

6 The Virtual Filesystem (VFS)

This topic is one of the most important, too. SimpleEdit isn't bound to the local filesystem at all. By leveraging a VFS we can offer support for any filesystem. It is, however, only implemented for the local filesystem and ftp-servers, currently. More implementations will follow.

I won't discuss the process of implementing the API here and I can't discuss every part of the user-API now, so I'll keep it short. The API consists of a few interfaces and a few classes. The interfaces are VFSEntry, VFSFile, VFSDirectory and VirtualFileSystem.

VFSEntry is the base interface for all entries that the filesystem contains. It defines convenient methods such as `getName()` and `toURI()`. VFSFile represents a file within a filesystem. Thus it provides special methods to access the contents of a file. It is important to remember that you have to cleanup after working with Streams from a file. For this purpose you'll have to do something like this:

```
InputStream is = null;
try {
    is = vfsFile.getInputStream();
    // do something with is
} catch(Exception e) {
    e.printStackTrace();
} finally {
    if(is != null) {
        try {
```

```
            is.close();
        } catch(Exception e) {
            e.printStackTrace();
        } finally {
            vfsFile.getFileSystem()
                .completePendingCommand();
        }
    }
}
```

This code is extremly ugly, thus we provide utilities for this task. As an example, the above code could be written as:

```
new ReadAccess(file) {
    protected void read(InputStream in)
            throws IOException {
        // do something with is
    }
};
```

You could have written it like this, too:

```
// define the class
public class MyReadAccess extends ReadAccess {
    protected void read(InputStream in)
            throws IOException {
        // do something with is
    }
}

// use it
ReadAccess ra = new MyReadAccess();
ra.read(file);
```

This enables reuse of readers. There's a WriteAccess as well.

The next interface we want to talk about is the VFSDirectory. This one represents directories on your filesystem. It provides methods to access child entries and some shortcuts for creation of files and directories.

VirtualFileSystem is the entry-point to any filesystem. It supports listeners on changes and some other neat stuff (like file/directory-creation). The methods in VFSDirectory are most likely just shortcuts with simpler syntax for the methods provided by this class.

How do you get one? That's what the FileSystem-class is good for.

```
VirtualFileSystem vfs = FileSystem.getFileSystem(
                "ftp://user:pass@host");
VFSEntry entry = vfs.getChild("/some/file.txt");
if(entry.exists() && (entry instanceof VFSFile)) {
    // ra is the ReadAccess we wrote earlier on
    ra.read((VFSFile)entry);
}
```

That's a pretty complete example, but it could've been written easier as FileSystem offers methods to directly retrieve an entry of a filesystem. Note that the above example would require a try-catch-block - in case your URI-format is wrong.

You can register your own filesystem to this FileSystem-class, too. Predefined formats are "local://", "file://" (both for the local system) and "ftp://" (for FTP, obviously).

When working with the VFS it is worth it to take a look at the VFSUtil-class as well. It provides methods for some tasks that can ease your work.

The VFS also has some utilities to implement a Visitor-Pattern (to keep this true: it isn't a "real" Visitor-Pattern, just an utility to make crawling a directory-tree easier). This is, how you'd use it, to visit all files in a specific directory (recursiv):

```
VFSEntry entry = LocalFileSystem.getInstance().getEntry(
            "/home/pago/tests");
VFSVisitor visitor = new RecursiveVisitor() {
    public void visit(VFSFile file) {
        System.out.println(file.getPath());
    }
};
visitor.visit(entry);
```

`RecursiveVisitor` extends the `AbstractVFSVisitor` (casts VFSEntry to VFSFile or VFSDirectory) which implements `VFSVisitor`. In case you want to write the paths of the directory visited as well, make sure you call super.visit(dir) as well, otherwise the recursive iteration won't work.

7 PgsAction - the action system

This topic would cover an entire document on its own, so I'll keep it as short as possible.

The PgsAction-package is a superset of the Swing-Action-API. The version we're using in SimpleEdit is even more advanced as it can use Java 5.0-features (namely annotations).

This document will be published together with an alpha release of PgsActions, thus I'll concentrate on the changes we did: In order to integrate the API tightly into SimpleEdit we had to write some implementations more specific. Instead of `com.pagosoft.action.AbstractSystemAction` you'll be using `org.simpleedit.action.SystemAction`. Instead of `com.pagosoft.action.ActionContainer` you'll use `org.simpleedit.action.SimpleActionCon` Instead of `com.pagosoft.action.AbstractStateAction` you'll use `org.simpleedit.action.SimpleAbstr` The other parts stay the same. The most important change is the resource-loading (which we might rewrite anyway thus rendering these classes obsolete, but for now: stay with them).

Want an example? Ok... here it goes:

```
// define action
public class MyAction extends SystemAction {
    public MyAction() {
        super("myaction");
    }

    public void actionPerformed(ActionEvent e) {
        // do something
    }
}

// add it to some menu
FileMenuAction fileMenu = ActionManager.getDefaultInstance()
            .getActionContainer(FileMenuAction.class);
fileMenu.add(MyAction.class);
```

Actions should be Singletons, don't forget that.

I said we're offering an additional API, so here we go:

```
// define the class
public class ActionHandler extends ActionObject {
    @ActionMethod(id="doSomething")
    public void doSomething() {
        // just do it
    }
}
```

```
// use it
ActionHandler handler = new ActionHandler();
JButton myButton = new JButton(handler.getAction("doSomething"));
```

It is highly discouraged to use this API if you want to develop an extensible part of SimpleEdit, but it's fine for dialogs and the like.

You don't need to extend ActionObject, but it would be easier for yourself. Instead you can use ActionFactory like this (assuming it does not extend ActionObject):

```
Map<String, Action> handler = ActionFactory.createActionMap(
                    new ActionHandler());
JButton myButton = new JButton(handler.get("doSomething"));
```

In this case, you'll have to keep a cache for yourself. If possible, use ActionObject.

8 To be continued

This document is not yet nearly complete. We've not even discussed the API for the Text-component, but right now you should have an impression on how the SimpleEdit-API works for you and where it does help you.

Later on there will be a Node-API that will ease the creation of Models for trees, lists, tables and comboboxes. This API already exists, but it has not yet been implemented for the GUI part. You can find it in the package `com.pagosoft.nodes`. Once it's finished, this code will move into `org.simpleedit.nodes` or will be moved into a separate package, offered by Pagosoft.