GtkEditor - The Next Generation

Design Document

Thomas Mailund

mailund@daimi.au.dk

Mark Slicker

jamess1@wwnet.com

GtkEditor - The Next Generation: Design Document

by Thomas Mailund and Mark Slicker

Revision History

Revision 0.1-Jan 27 2000-Revised by: mailund
Initial revision

Table of Contents

List of Figures

GtkEditor - The Next Generation

GtkEditor is a source code editor widget for GTK+ (http://www.gtk.org). The longterm goal of GtkEditor is to provide a framework for source code editing in GTK+ applications, whether just for editing macros or for full fledged IDEs. We try to achieve this through a highly configurable editor widget sub-classing the GtkText widget. After initialisation, the editor widget can be treated as the usual text widget, and all source-code editing features are taken care of by the widget itself. The behaviour of the editor is decided by dynamic libraries loaded during initialisation of the widget. These libraries can be hand-coded or created using tools supplied with GtkEditor.

The Next Generation (TNG) is the project name for the design and implementation taking place in the 0.1.x series. The 0.0.x series provided a flexible API for syntax highlighting making it applicable for a broad range of languages, but the API for other source editing facilities, such as automatic indenting, source browsing, and parenthesis matching ranged from poor to non-existing. The Next Generation is a total redesign and reimplementation, of the editor widget. The design and codebase from the 0.0.x series has been abandoned for what we consider a better and more flexible design.

This document describes the new design of the GtkEditor, *GtkEditor - The Next Generation*. Developers are encouraged to read through this document before diving into the source, and make sure to update it whenever they change the design or API. This document is considered canon on the GtkEditor design, and any design in the source *not* described here is considered wrong and is in danger of being removed.

1. Overview

The document is structured in two chapters. Chapter 1, describes the overall design of GtkEditor. It explains the different objects composing the editor design, describes their intended interaction, and how this interaction can be used to implement various editor features. This chapter should preferably be read by any developer before diving into the code. It is also recommended reading for any plugin developer. Chapter 2, lists the various structures and functions exported in the GtkEditor library. This chapter is intended as a reference manual for developers using the GtkEditor in their applications.

Chapter 1. Design of GtkEditor

1.1. Introduction

This chapter describes the design of GtkEditor. The main focus is on the architecture of the editor and on the objects composing it. We will stick to high-level descriptions of structures and algorithms, and leave people interested in low-level implementation details to the source code.

The chapter is structured in two sections, describing the main areas in the GtkEditor design. The first section, Section 1.2, describes the various objects found in the editor. The next section, Section 1.3 describes how the extension mechanism in the editor is structured, and describes the layout of plugins. Plugin developers can read this without reading the other sections, but are encouraged to skim the object section to get at least some idea about the general design.

1.2. The Objects

In this section we describe the various objects found in the TNG design, and the mechanism used for dynamic object loading. All objects, except for the actual editor widget, are dynamically loadable. The term *object* is perhaps a bit misleading, since in the TNG design, only the GtkEditor widget is an object in the GtkObject system, while scanners and plugins are not. The need for dynamic loading and unloading of scanners and plugins prevents us from implementing these two kinds of objects as GtkObjects. Instead we have a fixed interface that the loadable objects must implement, and the objects themselves are pure implementation.[1] All this, however, will be of little importance for the discussion in this paper, and we will make no distinction between the GTK+ objects and the loadable objects in the following.

FIXME: well...more of a mental note, really. I was thinking, we could have a scanner and plugin gtk object with load/unload class methods, so, e.g., a GtkEditorPlugin.new("objectfile"); woule create a new instance of of a plugin, initialised through the 'init' callback.

1.2.1. GtkEditor

The editor widget is a subclass of GtkText. GtkEditor does little in the way of extending the text widget in it self, but it provides a mechanism for installing plugins responsible for this, as well as a framework for lexical editing (as opposed to character editing).

The three major attributes of the editor widget are the *scanner*, the *plugins*, and the *text_properties* table. The scanner, described in detail in Section 1.2.2, is responsible for lexical analysis of the source being edited. The scanner performs incremental lexical analysis to keep a list of tokens updated, and it provides signals when matching certain tokens, such as, e.g., comments, keywords, functions, or newlines. Scanners are, in nature, language specific, and the specific signals and the matched tokens depends on the connected scanner. A scanner can be connected or disconnected at any time. If a new scanner is connected while another is still connected, the old scanner will automatically be disconnected. When a scanner is disconnected, all connected plugins will also be disconnected.

The *plugins*, described in detail in Section 1.2.3, provides the mechanism for language specific extensions of the widget. An editor can contain any number of plugins, which can be connected or disconnected at any time. In the current design highlighting and indenting will be performed by plugins. There are no distinction between different kinds of plugins, so it is quit possible to install, say, two indenters at the same time, with unpredictable consequences. We might want to change that in the future, but right now the flexible design, where the user is assumed to be competent, is preferable to the more secure design, where the user is inhibited.

Finally, the *text_properties* table is used for syntax highlighting, or more precisely, for determining the text properties, such as font and colours, of inserted text. This is described shortly in Section 1.2.1.1

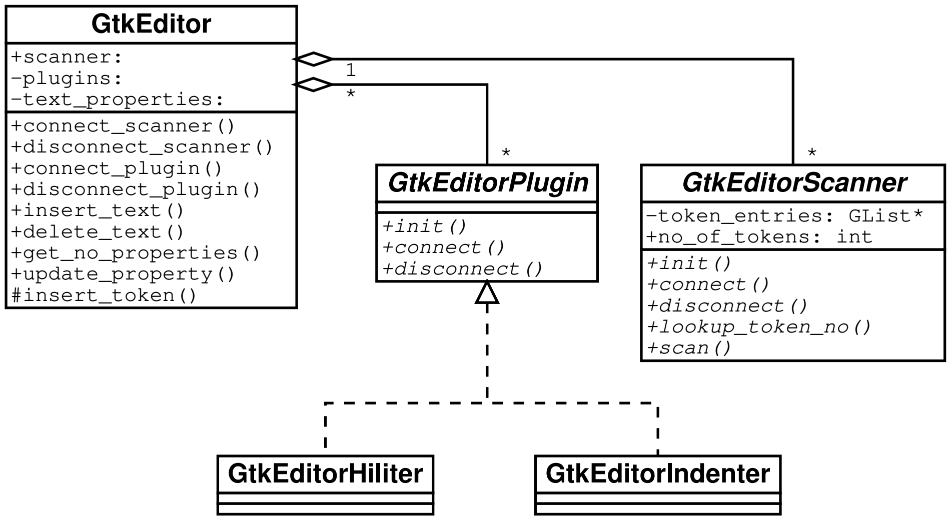
Figure 1-1 shows a UML diagram of the GtkEditor architecture. The class `GtkEditor` contains the public attribute *scanner*, and the two private attributes *plugins* and *text_properties*. The attribute scanner is an object of type `GtkEditorScanner`. The attribute *plugins* is a collection of objects of type `GtkEditorPlugin`. Finally, the attribute *text_properties* is a table of text properties. This table is described in Section 1.2.1.1. The class `GtkEditor` has methods for connecting and disconnectiong scanners and plugins, for inserting and deleting text, and for updating text properties and inserting tokens. The later three are also described in Section 1.2.1.1.

Of the attributes, only the *scanner* is public. This should not be misused by updating scanners through anything but the two methods `connect_scanner` and `disconnect_scanner`. The scanner, however, can safely be accessed through its methods. The attributes *plugins* and *text_properties* should only be accessed through the appropriate editor methods.

All methods, except for `insert_token` are public. The method `insert_token` is used by the scanner to insert new tokens found during lexical analysis, and should only be used by the scanner.

Both the scanner class and the plugin calls are really just interfaces, that dynamically loaded libraries must implement to be used in these roles. In the current design, functionality such as highlighting and indenting will be implemented by plugins.

Figure 1-1. GtkEditor Architecture



1.2.1.1. Highlighting

The scanner is responsible for lexical analysis of the text, the editor is responsible for inserting/deleting text, and the highlighter is responsible for the way the text is displayed, by setting the text properties. The mechanism is the following: the editor has a table, *text_properties,* indexed by token numbers. For each token, the corresponding entry in the properties table describes how the token should be displayed. The scanner manages the tokens, and thus determines both the number of tokens, and the mapping from tokens to token numbers. When the scanner is connected to the editor, the editor probes the scanner for the number of tokens and allocates the *text_properties* table.

Initially, all entries in the properties table is empty, meaning the corresponding tokens should be inserted with the default properties. Plugins can change this when they are connected, using the editor method `update_property`, and in this way determine the syntax highlighting. Token numbers is not the right abstraction for highlighting, since they will change between different scanners, and even different versions of the same scanner. For this reason, the scanner implements a mapping, `lookup_token_no`, from token names, which are strings such as "comment" and "keyword", to token numbers. This mapping is used by the editor for updating the properties table. An implementation of `update_propery` could look like this:

```
int token_no = scanner.lookup_token_no (token_name);
if (text_properties[token_no])
```

```
free_property (text_properties[token_no]);
text_properties[token_no] = token_props;
```

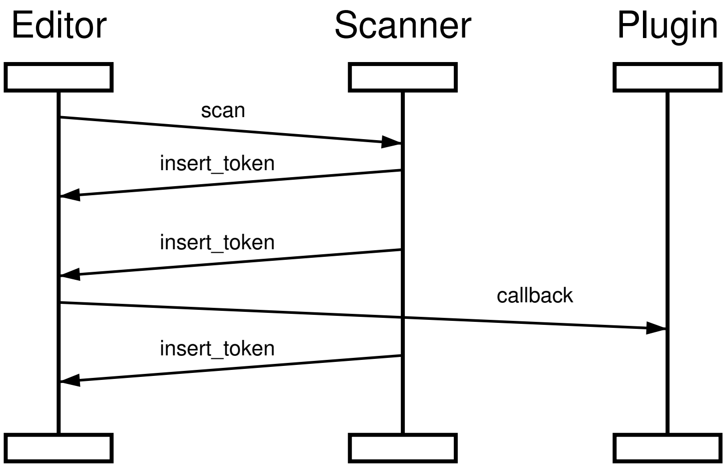
The entries in the properties table can either be text properties, i.e. font, fore- and back-colour, or it can be a callback of the type

```
void (*callback)(int position, const char *token, int token_size)
```

If the entry in the property table consists of text properties, associated tokens will be displayed with these properties. If the entry consists of a callback, this function will be called. The callback is then responsible for inserting the token with the appropriate text properties. *FIXME: we need to find out how to avoid infinite recursion here.*

When text is inserted into the editor widget, the method `insert_text` is invoked. This method calls the scanner method `scan` for lexical analysis of the new text. For each token the scanner finds, it calls the `insert_token` method in the editor. The editor then looks up the text properties in the *text_properties* table and either insert the text with the text properties from the table entry, or calls the callback in the entry.

Figure 1-2. Insert text



It can be useful to use more than one plugin for highlighting. One could, for example, use a generic plugin for setting text properties for common tokens such as comments, strings, and keywords, and then combine this plugin with a highlighter that only sets the properties for prototypes, for use with a C scanner, or combine the same generic highlighter with a highlighter that sets properties for module constructions such as "functor", "signature", or "structure", for use with an SML scanner. Notice, however, that if more than one plugin modifies the properties table, the order of plugin connections determines the actual syntax highlighting, since later connections will overwrite updates caused by earlier connections.

1.2.1.2. Indenting

FIXME: Nothing here yet!

1.2.2. GtkEditorScanner

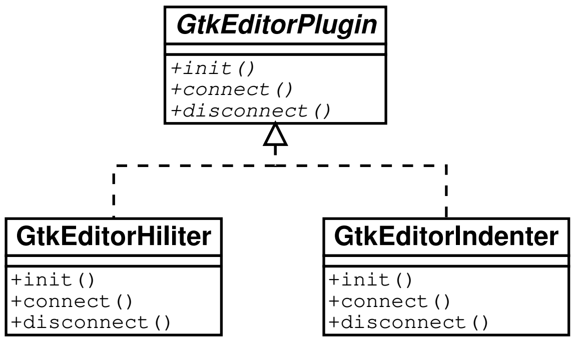
Scanners are responsible for the lexical analysis of the source code and for providing the mechanism for editing on token level rather than character level. The scanner performs incremental lexical analysis to keep a list of tokens updated, and it provides signals when matching certain tokens, such as, e.g., comments, keywords, functions, or newlines. Scanners are, in nature, language specific, and the specific signals and the matched tokens depends on the connected scanner.

FIXME: more stuff here. We probably want a description of the scanner we use right now. Not as a fixed algorithm, but as one way (our way) of implementing a scanner.

1.2.3. GtkEditorPlugin

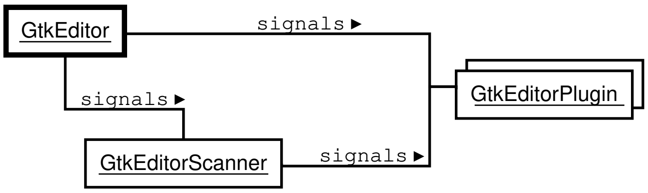
The plugins provides the mechanism for language specific extensions of the widget. The plugin described here is an interface, that objects must implement to be used as plugins. In the current design both highlighting and indenting is managed by plugins.

Figure 1-3. The Plugin Hierarchy



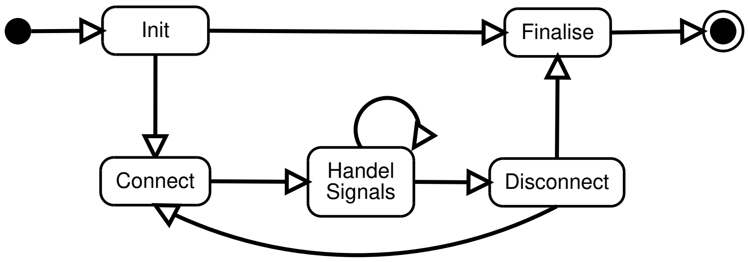
Plugins are objects that are connected to editors, where they sit passive and listen for signals from either editor or scanner.

Figure 1-4. The Plugins



Plugins can be loaded from file as dynamically linked libraries. When a plugin has been loaded it is initialised through its method `init`. This method can be used to setup structures shared by all connections. When a plugin is connected to an editor, its `connect` method is called. This method is used to connect the plugin to signals in the editor and scanner, and is used to setup connection specific structures. When a plugin is disconnected, another method, `disconnect`, is called. This method should be used to free up resources and to disconnect from any signals in the editor or associated scanner. Before a plugin is unloaded, the method `finalise` is invoked. This can be used to free up any resources allocated during initialisation.

Figure 1-5. The Life of a Plugin



FIXME: We need a refcount in plugins

1.3. The Plugin Mechanism

FIXME: nothing here yet! We should describe the mechanism hintet at in the description of GtkEditor

Notes

1. If we find that we need a full fledged object system for our plugin mechanism, we will implement it. One possibility is to base it on the BSE object system, which is similar to the GtkObject system, but which has support for dynamic loading and unloading of objects. We have not yet found the need to do this.

Chapter 2. The Library API

FIXME: VERY in-complete. I think we want to generate this chapter from our source, using gnome-doc or similar.