My Favorite Editor Anywhere

Hayco de Jong and Taeke Kooiker

CWI, Department of Software Engineering
Kruislaan 413, 1098 SJ Amsterdam, The Netherlands
(jong|kooiker)@cwi.nl

Abstract. How can off-the-shelf editors be reused in applications that need mature editing support? We describe our editor multiplexer which enables interactive, application guided editing sessions using GNU Emacs and Vim. At a cost of less than 1 KLOC of editor-specific *glue* code, both IDE builders and users benefit. Rapid integration of existing editors reduces application development cost, and users are not confronted with yet another foreign editor with its own learning curve.

1 Introduction

Many applications such as email clients, instant messengers, web browsers, and programming environments provide editing facilities. Full fletched, off-the-shelf editing solutions such as GNU Emacs [1] and Vim [2] are readily available, but many application developers still choose to write their own editing software. Some utility libraries (e.g. Java's JFC/Swing library) contain partial solutions in the form of reusable editing widgets. Still, developing and extending your own editor to encompass the feature richness common in mature text editors is far from a *rapid* software engineering exercise.

Offering a single built-in editor obviously also limits the user to this editor. This poses no problem as long as the editing sessions are brief, e.g. during login or password entry. However, when the editor is used for lengthy (programming) sessions, being forced to use the keybindings dictated by an editor that is not your personal favorite can easily lead to frustration.

This paper describes how we reuse and integrate existing editors in a programming environment. Although our implementation is based on needs we have in our own environment, both the idea and most of the implementation can carry over to other projects. Basically, projects that need editing support for structured documents and where interactivity with these editing sessions is desirable, could benefi t from the architecture we describe.

The structure of this paper is as follows. This section continues with some background, motivation and discussion of related work. Section 2 describes how we coordinate simultaneous editing sessions, and we show the architecture used to deal with various editors. Section 3 describes some of the implementation details of the architecture: the MULTIPLEXER which orchestrates simultaneous editing sessions and the *glue* that is needed between the MULTIPLEXER and the various editor instances. We conclude with a summary of our contribution and a discussion of ideas for future work in Section 4.

1.1 Background

The ASF+SDF Meta-Environment [3,4] is a programming environment generator: given a language definition consisting of a syntax definition (grammar) and tool descriptions (using rewrite rules) a language specific environment is generated. Figure 1 shows a screenshot of the ASF+SDF Meta-Environment. A language definition typically includes such features as pretty printing, type checking, analysis, transformation and execution of programs in the target language. The ASF+SDF Meta-Environment is used to create tools for domain-specific languages and for the analysis and transformation of software systems.

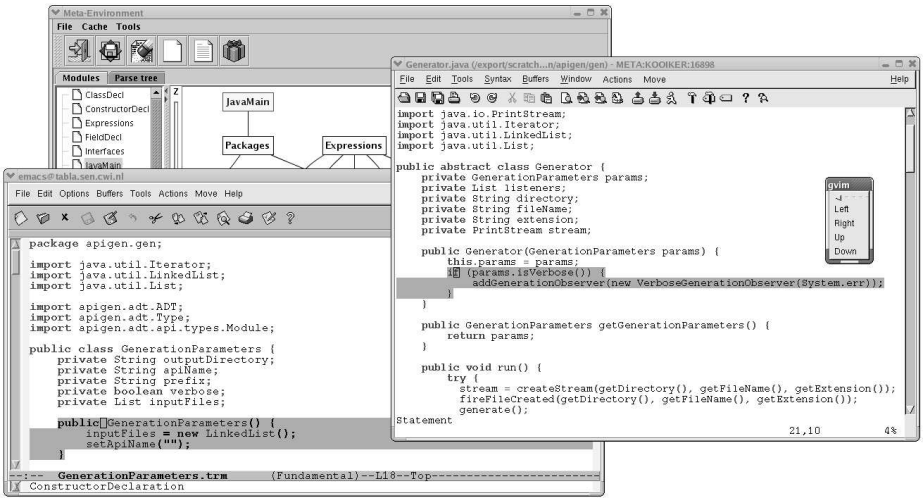

Fig. 1. GNU Emacs and Vim used simultaneously by an IDE.

The ASF+SDF Meta-Environment is used in several academic [5], industrial [6], and financial projects [7,8]. Presently, the ASF+SDF Meta-Environment is intensively used in the software renovation oriented research project CaLCE: "Computer-Aided Life Cycle Enabling". This project is financed by the Dutch Ministry of Economic Affairs and aims at the development of tooling to improve the overall quality of systems deployed in the financial setting.

1.2 Related work

Some applications (e.g. the KDE and Gnome window managers) allow the configuration of a *foreign* editor. Whenever a body of text needs to be edited, the application executes the configured editor and waits for the user to complete the editing session. During this session, there is no interaction between the main application and the foreign editor: the editing session is *unguided*. In some applications instantiations of external editors can be *embedded*. Some examples are KDE's filemanager `konqueror`

and email reader `kmail` which can embed instances of a specially crafted version of the Vim text editor. In these cases, the host application (`kmail`) encapsulates the editor (`kvim`) and shows its window as if the editor were part of the application. This gives the user the feeling that his favorite editor is integrated in the application, even when this integration is only visual and there is no real interaction between host application and editor.

Our focus is not so much on the *visual* integration achieved by embedding the editor instances. Instead we emphasize *functional* interaction *during* the editing session.

Another way to look at application-editor interaction is to look at the editor as the main application, and to view external tools as subordinates of the editor. Especially users of the Emacs family of editors find ways to link their email reader, spell-checker, or other popular application into Emacs by writing support *glue* in Emacs LISP.

2 Design

In any IDE it is common to have multiple simultaneous editing sessions, as users start and finish editing, switching from one file to another. To take care of any administrative issues we have to deal with the following tasks:

Managing Using multiple editing sessions requires administration of open sessions and addressing these editing sessions.

Executing Supporting several editors almost certainly results in different startup procedures for each editor. We provide an open and generic architecture for supporting several editors.

Marshalling We need full interaction with the supported editors, which means that data has to be transferred from the application to the editor instance and vice versa.

We first have a look at the requirements (Section 2.1) and then split the design into editor-independent (Section 2.2) and editor-specific (Section 2.3) details, and we show how the components connect (Section 2.4) to form our multiplexing editor architecture.

2.1 Requirements and considerations

Given our experience with editing issues in the Meta-Environment (Section 1.1) we are interested in a solution which is:

Noninvasive We are strongly determined *not* to edit the source code of any particular editor itself.

Simple Keep the number of methods in the editor interface low: 10 rather than 100 methods. Prefer implementation of these methods in established programming languages (e.g. C or Java), rather than the editor's (sometimes arcane) domain specific scripting language.

Open Both in terms of *platform* and *language*:

- Platform independence: although designed for a Unix environment, the implementation should be independent of whether this is e.g. Linux, SunOS, or Windows/Cygwin;

– Language independence: the architecture does not dictate any particular programming language for the editor connectors.

From the Meta-Environment point of view, we are at least interested in the following interesting editor actions and events:

Menu We want to add menu items in the editor which, when selected by the user, are forwarded to the environment where they are handled.

Cursor Cursor positioning and text highlighting can be directed by the environment (model) and rendered in the editor (view).

Modification The editor notifies the environment of any changes the user makes to the file.

Save/Load The environment can request the editor to save its contents or re-read them from the file system.

We start out with this restricted set, but we keep the design open to allow for later extensions. The less demands, the more editors we can potentially support. If for example an editor offers no support to add user-defined menus, we cannot set them up from another application either. Although we *could* patch the editor sources to add menu support we deliberately refrain from doing so.

2.2 Editor-independent design

The editor-independent design describes a generic way of managing and communicating with editor instances. Without knowledge of the actual editor instance, one can provide an abstract level of communication by defining a common interface which provides all necessary functionality to fulfill the requirements given in Section 2.1. A tool that implements this design takes care of managing editing sessions, including starting and shutting down sessions, and communication with these editing sessions. The MULTIPLEXER described in Section 3.1 is a tool that implements this.

2.3 Editor-specific design

Managing editing sessions can be done in a generic way, but actual communication and execution of editor instances has to be editor specific. This communication can be done in various ways. While Vim makes use of an arcane syntax-based communication protocol via the commandline, OpenOffice for example can be controlled by using an extensive API. These differences lead to different design implementations for different editors. To prevent changes to the MULTIPLEXER for every editor that has to be supported we introduce a connector (see Section 3.2) mechanism which separates communication with the actual editor instances from managing the editing sessions. For each supported editor there has to be a corresponding connector. All editor-specific communication details are known to this connector, while the MULTIPLEXER can be implemented in a generic way. The generic interface provided by the MULTIPLEXER has to be implemented by every connector.

2.4 Execution models

No two editor implementations are the same, and they are often written based on different designs. Editors based on the GNU Emacs philosophy prefer to interact with external processes only if they are executed by the editor. Other editors are more easily controlled by an external process.

We accomodate for this difference by allowing two execution models. Either the MULTIPLEXER first launches the connector which launches the editor, or the MULTIPLEXER launches the editor instructing it to immediately launch the connector.

Independent of the execution model, the final state is the same: the MULTIPLEXER communicates with the editor via a dedicated connector (Figure 2).

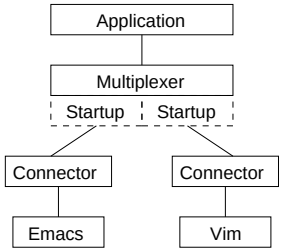

Fig. 2. Overview showing how editors are connected to an application.

3 Implementation

Given the design from Section 2, we describe the MULTIPLEXER which contains the editor-independent implementation in Section 3.1. This MULTIPLEXER invokes interface methods which in turn are implemented in editor-specific connectors which are detailed in Section 3.2. Finally, we explain how we *glue* it all together in Section 3.3.

3.1 Editor Multiplexer

The editor MULTIPLEXER manages multiple simultaneous edit sessions by assigning each session a unique ID. Subsequent calls to the edit session carry this ID as one of the call's parameters. This allows the MULTIPLEXER to uniquely identify to which connected editor the request needs to be forwarded.

The MULTIPLEXER is currently implemented as a TOOLBUS tool, written in the `C` programming language. The TOOLBUS coordination architecture is a middleware layer with a process algebra based scripting language. [9] offers a comprehensive explanation of the TOOLBUS scripting language. Because the entire Meta-Environment architecture uses the TOOLBUS coordination architecture, making the MULTIPLEXER a TOOLBUS tool is the obvious choice. For applications that do not use the TOOLBUS, an implementation in the form of a `C` library would be equally feasible.

The choice for C as the implementation language was pragmatic. C offers direct access to operating system functionality such as process duplication through the use of the fork system call, execution of external processes using exec and has additional low level support for sockets, pipes and file descriptors. Although we also experimented with an implementation in Java during research in the context of connecting the Eclipse IDE editor [10], we opted for C's easy link to operating system functionality.

We show a simplified ToolBus interface definition of our Multiplexer.

```
01  tool multiplexer is { command = "./editor-multiplexer" }
02
03  process EditorMultiplexer is
04  let
05    EM: multiplexer,
06    Editor, Filename: str,
07    SessionID, SL, SC, EL, EC: int,
08    MainMenu, SubMenu: str
09  in
10    execute(multiplexer, EM?)
11    .
12    (
13      rec-msg(edit-text(Editor?, Filename?))
14      . snd-eval(EM, Editor, Filename))
15      . rec-value(EM, SessionID?)
16      . snd-msg(edit-text(Editor, Filename, SessionID))
17    +
18      rec-msg(set-focus(SessionID?, SL?, SC?, EL?, EC?))
19      . snd-do(EM, set-focus(SessionID, SL, SC, EL, EC))
20    +
21      rec-event(EM, menu-selected(SessionID?, MainMenu?, SubMenu?))
22      . snd-msg(menu-selected(SessionID, MainMenu, SubMenu))
23    )
24    * delta
25  endlet
```

This example is limited to showing the execution (line 10) of the previously declared multiplexer tool (line 01). Following the execution is a looping construct (lines 12-24). During each iteration exactly one of the declared scenarios can occur. First, a request to start a new session is handled (lines 13-16). Second a request to set the focus to a particular region delimited by start-line, start-column, end-line and end-column (lines 18-19) to any existing editor can be handled. Finally, a menu event can come in from one of the connected editors (lines 21-22).

Applications that do not use the ToolBus, could use e.g. pipes, sockets or library calls to communicate with the Multiplexer.

3.2 Editor Connectors

For each supported editor, we implement a small connector that translates the editor-independent interface calls into the editor specific implementation. These connectors are necessary because each editor has its own unique scripting facilities or programming language (Vim uses Vim script, GNU Emacs uses Emacs Lisp), and because communication with each editor is usually handled in a slightly different way. We describe the connectors we implemented for Vim, GNU Emacs, and for a proprietary implementation of an editor in JFC/Swing.

Vim The Vim connector is implemented partially in `C` and partially in Vim's scripting language. The `C` functions implement the text editor interface. Commands *from* the MULTIPLEXER *to* the editor are sent using Vim's remote scripting feature.

For example, the implementation of the `setCursor(int offset)` method looks like this:

```
01  static void gotoCursorAtOffset(int offset) {
02    char cmd[BUFSIZ];
03    sprintf(cmd, ":goto %d", offset);
04    sendToVim(cmd);
05  }
```

Events *from* the editor *to* the MULTIPLEXER, are initiated by Vim. E.g. Vim is instructed to forward buffer changes resulting from user editing by means of the Vim hook called `BufWritePost`:

```
01  func! EnableModificationDetection()
02    autocmd BufWritePost * :call BufModified()
03  endfunc
```

where `BufModified` is a function (in Vim script) that forwards this event to the MULTIPLEXER.

Currently, the editor-specific glue for Vim is expressed in 501 lines of `C` code, and 77 lines of Vim script.

GNU Emacs Similar to the `sendToVim` function, `sendToEmacs` is used to communicate from the MULTIPLEXER to GNU Emacs. The difference is that where Vim lacks a regular communication channel and we had to resort to using its remote scripting feature, with GNU Emacs we can communicate using a a pipe.

```
01  static void sendToEmacs(int write_to_editor_fd, const char *cmd) {
02    write(write_to_editor_fd, cmd, strlen(cmd));
03    write(write_to_editor_fd, "\n", 1);
04  }
```

The communication channel may be simpler in this version, but not all comes easy when dealing with GNU Emacs. The initial scripting necessary to setup the connector is programmed in Emacs LISP:

```
01  (defun init (args)
02    (setq emacs-connector
03      (let ((process-connection-type nil))
04           (apply 'start-process "emacs-connector" "*Meta*" "emacs-connector"
05             (split-string args))))
06    (set-process-filter emacs-connector 'multiplexer-input)
07    (process-kill-without-query emacs-connector)
08    (define-key global-map [mouse-1] 'mouse-clicked)
09    (add-hook 'after-change-functions 'buffer-modified () t)
10  )
```

Lines `02-08` execute the connector and register the LISP function `multiplexer-input` as input handler for the connector. Line `10` registers a mouse-click listener, and line `11` registers the `buffer-modified` function so it gets invoked whenever user editing causes the buffer to change.

Currently, the editor-specific glue for GNU Emacs is expressed in 436 lines of `C` code, and 108 lines of Emacs LISP.

JFC/Swing Editor As an experiment and possible extension to the ASF+SDF Meta-Environment, we also created an editor based on the GUI classes available in JFC/Swing. Again similar to the previous implementations, we were able to connect this Java editor to the MULTIPLEXER. We do not show implementation details, but it is worthwhile to mention that the connection to this editor is based on sockets, rather than pipes (as we used for the GNU Emacs connector). Although we could have used the commonly accepted route where the standard input and output streams are sacrificed and used for communication via a pipe, we opted for the socket approach, just to add this route to our repertoire.

Currently, the editor-specific glue for our JFC/Swing editor is expressed in 411 lines of C code, and a 5 line shell script to invoke java with the correct classpath for the editor.

3.3 Glueing it all together

Now that we have the editor-independent MULTIPLEXER, and the editor specific connectors, we can finally glue them together to get a working system. We describe how the MULTIPLEXER executes and communicates with an editor.

Executing an editor The MULTIPLEXER executes the requested editor as follows. For each editor, we write a small piece of (C) code that is loaded as a dynamic library. This mini library contains a single `startup` function with three parameters: the filename to be edited and the two file descriptors to be used for communication with the MULTIPLEXER. The startup function for the Vim editor looks like this:

```
01  void startup(const char *filename, int readFromFD, int writeToFD) {
02    char fromMultiFD[10], toMultiFD[10]; /* file descriptors as string */
03
04    sprintf(fromMultiFD, "%d", readFromFD);
05    sprintf(toMultiFD, "%d", writeToFD);
06
07    execlp("gvim-connector", "gvim-connector",
08           "--read_from_multiplexer_fd", fromMultiFD,
09           "--write_to_multiplexer_fd", toMultiFD,
10           "--filename", filename,
11           NULL);
12
13    perror("execlp:gvim/startup");
14    exit(errno);
15  }
```

The MULTIPLEXER invokes `startup` by using the `dlopen` and `dlsym` system calls (not shown here) for interacting with dynamic libraries. We thus *extend* the MULTIPLEXER with a single function per specific editor.

In the `startup` function, we choose one of the two execution models described in Section 2.4. For Vim we `exec`ute (lines `07-11`) the connector, thus following the *connector first* execution model.

For GNU Emacs, we have a similar `startup` function. Only it was more convenient to execute `emacs` first and have it fire up the connector instead. GNU Emacs is then told to load the editor-specific startup script (in this case written in Emacs LISP) and to begin by executing the function `init`:

```
01  void startup(const char *filename, int readFromFD, int writeToFD) {
02    char evalargs[BUFSIZ];
03    sprintf(evalargs,
04            "(init \"--read_from_fd %d --write_to_fd %d --filename %s\")",
05            readFromFD, writeToFD, filename);
06
07    execlp(EDITOR, EDITOR, filename, "-load", "gnu-emacs.el",
08           "-eval", evalargs, NULL);
...   /* error handling code omitted */
11    }
```

Communicating with an editor Depending on the functionality offered by each specific editor, we use different means of setting up a communication channel with the editor. We have used different channels ranging from a `pipe` (in GNU Emacs), to a `socket` (in the JFC/Swing editor), to the more esoteric remote scripting feature offered by Vim.

Independent of the type of the available communication channel, we use the *same* technique to *marshal* data over this channel. Instead of writing ad-hoc marshalling and de-marshalling code in the MULTIPLEXER and the connectors, we use APIGEN [11]. APIGEN takes as input an abstract data type description (ADT) and generates a `C` library or `Java` jar-file containing a.o. `set`, `get` and serialization methods.

Each command to and event from the editor is formalized in the text editor ADT. From this specification APIGEN generates the API implementation which we use to (de-)marshal communication between the MULTIPLEXER and editor.

4 Discussion and Future work

We have implemented a framework that allows reuse of off-the-shelf editors such as GNU Emacs and Vim in the ASF+SDF Meta-Environment. By implementing as much as possible of this framework in a generic, editor-independent way (our MULTIPLEXER), we can easily and rapidly add other editors to our environment. Deploying code generation techniques (APIGEN), and an available (programmable) middleware layer (TOOLBUS) ensures the solution is cheap in maintenance.

Our editing solution is *noninvase*: we never change any editor internals, *simple*: only a handful of lines of code in the editor's own scripting language are needed, and *open*: our editing support has been tested on various Linux platforms, and we have both `C` and `Java` connectors.

Our editing framework was primarly designed for use in the Meta-Environment, which relies heavily on the TOOLBUS as its middleware layer. However, our contribution is not limited to using the TOOLBUS, and we plan to offer our results for use in a non-TOOLBUS setting, as a downloadable package.

Another direction of interest is figuring out in which ways we can expand the text editor interaction. We have already experimented with syntax highlighting (i.e. one tool describes which part of the text gets which font attributes and colour and the rendering is done by the text editor), and structured editing, but conceivably several more applications can benefit from our support.

Obviously, the more complicated the things we demand, the fewer editors we will be able to fully support. Vim, for example lacks atomic functionality to colour a specific

region of characters, although it does offer complex syntax highlighting. This leads to the following question: *what is the set of text editing primitives small enough to be covered by almost any editor, but large enough to be useful in most applications that require editing?*

Finally, as its name states, one of the MULTIPLEXER's task is *multiplexing* simultaneous editing sessions. In a coordination architecture such as the TOOLBUS, the multiplexing *concern* could be applicable to other tools as well. If this notion were lifted to a TOOLBUS primitive, any setting that launches multiple instances of a tool with the same interface could possibly benefi t.

References

1. Stallman, R.M.: Emacs the extensible, customizable self-documenting display editor. In: Proceedings of the ACM SIGPLAN SIGOA symposium on Text manipulation. (1981) 147–156
2. Moolenaar, B.: Vim is a highly configurable text editor built to enable efficient text editing. Vim 6.3 is available for download from `http://www.vim.org` (2004)
3. van den Brand, M.G.J., van Deursen, A., Heering, J., de Jong, H.A., de Jonge, M., Kuipers, T., Klint, P., Olivier, P.A., Scheerder, J., Vinju, J.J., Visser, E., Visser, J.: The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In Wilhelm, R., ed.: Compiler Construction (CC '01). Volume 2027 of Lecture Notes in Computer Science., Springer-Verlag (2001) 365–370
4. Klint, P.: A meta-environment for generating programming environments. ACM Transactions on Software Engineering and Methodology **2** (1993) 176–201
5. van den Brand, M.G.J., Iversen, J., Mosses, P.D.: An Action Environment. In: Electronic Notes in Theoretical Computer Science, Elsevier (2004) to appear.
6. van den Brand, M.G.J., van Deursen, A., Klint, P., Klusener, S., van der Meulen, E.A.: Industrial applications of ASF+SDF. In Wirsing, M., Nivat, M., eds.: Algebraic Methodology and Software Technology (AMAST'96). Volume 1101 of Lecture Notes in Computer Science., Springer-Verlag (1996) 9–18
7. Klusener, S., Lämmel, R.: Deriving tolerant grammars from a base-line grammar. In: Proceedings of the International Conference on Software Maintenance, IEEE Computer Society (2003) 179–189
8. Veerman, N.P.: Revitalizing modifiability of legacy assets. Journal of Software Maintenance and Evolution: Research and Practice **16** (2004) 219–254
9. Bergstra, J., Klint, P.: The discrete time ToolBus – a software coordination architecture. Science of Computer Programming **31** (1998) 205–229
10. van den Brand, M.G.J., de Jong, H.A., Klint, P., Kooiker, A.T.: A language development environment for eclipse. In: Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange, ACM Press (2003) 55–59
11. de Jong, H.A., Olivier, P.A.: Generation of abstract programming interfaces from syntax definitions. Journal of Logic and Algebraic Programming (JLAP) **59** (2004) 35–61 Issues 1–2.