Version Control in EVE

Software Engineering Laboratory

By: Emanuele Rudel
Supervised by: Nadia Polikarpova
Prof. Dr. Bertrand Meyer

Student number: 08-927-758

Contents

1. Introduction

1.1 Motivation

The Eiffel Verification Environment (EVE) is a large project that makes extensive use of Apache Subversion. In the current situation, the developer brings changes to the code first, and then relies on an external tool to interact with the version control system. The former step is performed with EiffelStudio or EVE itself, while for the latter step the command line, operating system extensions, like Tortoise SVN, or third party software are used. The whole process works flawlessly, but it is not efficient nor seamless since it requires the user to switch back and forth between different programs.

1.2 Goal

The goal is to integrate a Subversion tool within EVE in such a way that the user only has to interact with one program, namely the IDE. The benefits from the user's point of view are the increase of efficiency and above all the reduced amount of time required to learn new tools.

1.3 Outline

In order to provide the tool described above, it is necessary to bridge the gap between the Subversion client and Eiffel: section two explains in detail how this is accomplished. The third and fourth sections will cover the development of the actual tool for EVE from a technical point of view.
The purpose of the attached document *Version Control in EVE: User Manual* is to show the user how to use the graphical tool in EVE. A brief tutorial on how to use the Subversion library is also provided.

2. Subversion Client Library

2.1 Problem Statement

The Subversion client library for Eiffel is not directly related to the project in the sense that it is not essential to implement the tools illustrated in the next sections, however it helps separating different concerns and is designed for reusability. It allows thus other developers to take advantage of Subversion by using this library – as explained in the aforementioned user manual – and to possibly extend it by adding new commands.

2.2 Architecture

2.2.1 Commands

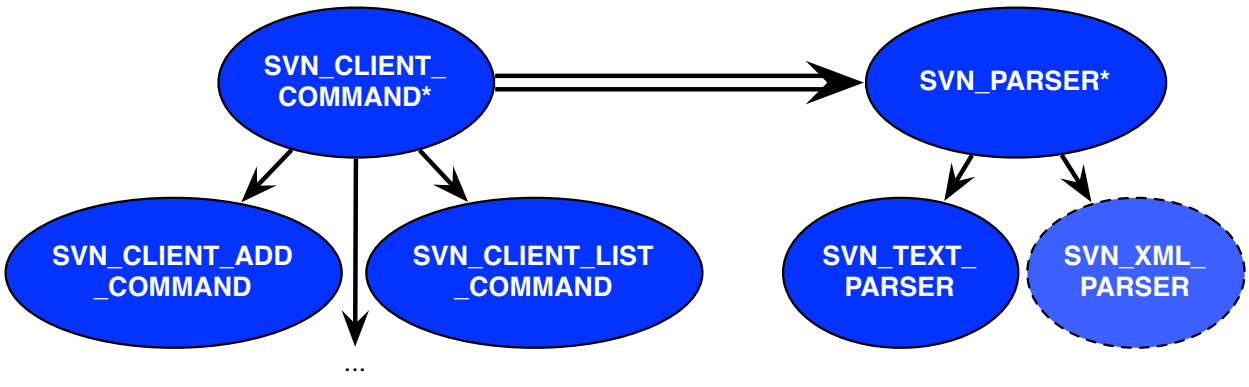
Subversion client commands are very similar to each other in how they are executed: they usually require one input argument (also known as the *target*) and a variable number of optional arguments. The output result is returned as plaintext, regardless of whether the execution was successful or not. Thus, they all share some common behavior defined in the deferred class SVN_CLIENT_COMMAND. Some commands may require two arguments (*source* and *target*) and again a variable number

of options. The following (intentionally incomplete) class diagram illustrates the relevant features of an abstract command.

SVN_CLIENT_COMMAND
make (a_svn_client: SVN_CLIENT)
execute
target: STRING last_result: STRING last_error: STRING
set_target (a_target: STRING) put_option (a_option_name, a_value: STRING)
{NONE} *parse_result (a_svn_parser: SVN_PARSER) *command_name: STRING

Each concrete command has a unique name and can parse the output result, although it is not always necessary. The execute feature collects the target and options, if any, and asynchronously executes the actual Subversion command. All the commands are non-blocking as they might slow down the whole workflow otherwise. A client can register to three callbacks of a command that is being executed, namely on_error_occurred, on_finish_command and on_data_received. The second callback is called only if the first one isn't and vice-versa, i.e. the command has either completed successfully or not, while the third callback is fired every time a partial result (could be a line of a long repository's checkout) is received. Please note that one has to register to the callbacks before invoking the execute command.

Once the execution terminates, the last result and error are updated and possibly parsed into a different format. The parsing process takes place in a descendant of the SVN_PARSER class in order to separate the result from its representation. What we obtain is the so called visitor pattern, depicted below in a slightly simplified class diagram.



Developers can thus extend the commands in the library by just adding a descendant class of SVN_CLIENT_COMMAND and defining the command name. New parsers, like a XML one for example, can be added without changing a line of code in

the commands classes: it is only needed to add a descendant of SVN_PARSER and implement the deferred features declared in it.

2.2.2 Client

The main purpose of the SVN_CLIENT class is to group all the commands in one place so that the user can easily access them. Likewise, for each command available in the client there's a corresponding list of supported options defined by the *cmd*_list feature, where *cmd* is to be replaced with the actual Subversion command. This class also defines the directory in which the commands should be executed (*working path*) and allows the user to select different parsers.

2.2.3 Parser

The parser included in the first version of the library only processes the simple plaintext output by the Subversion command line client, i.e. does not parse commands that use the verbose option or similar. Moreover, some commands also support the XML output option and hence an appropriate parser could be developed in the future.

3. Repositories Tool

The Repositories tool allows the user to inspect and to checkout repositories (or a part of them).

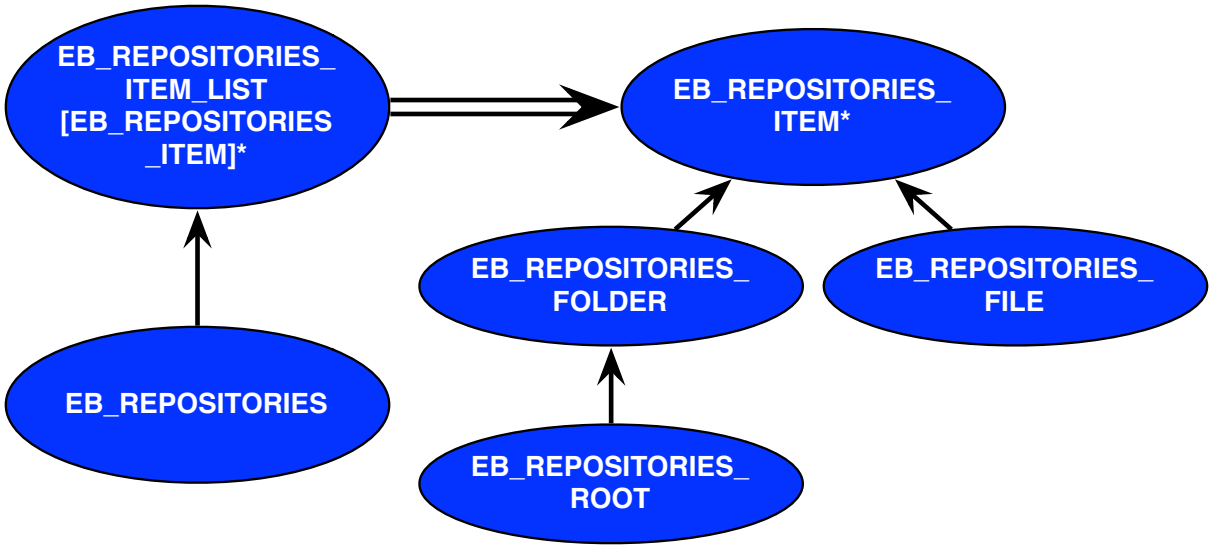
3.1 Architecture

The repositories are modeled with tree structures which are mapped to their respective graphical tree items. Each tree item of the model holds the name of its corresponding repository item and a list of sub items, if it is a folder. There's one root item for every repository that holds the repository URL and is responsible for listing its content. At the time of writing, only the checkout Subversion command is available to the Repositories tool, although it is discussed in the subsection *Future Work* how more commands can be added.

As illustrated in the figure below, the repository model is entirely managed by the EB_REPOSITORIES class. The graphical representation of the repositories is handled by the EB_REPOSITORIES_TREE class, which is always consistent with the model thanks to a publisher-subscriber pattern that notifies the model's changes to the view calling the features on_item_added, on_item_removed and on_update.

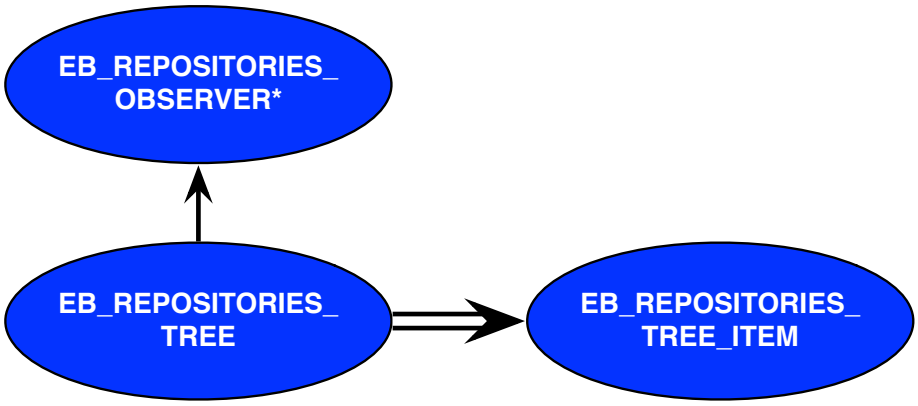
The EB_REPOSITORIES_MANAGER class is a façade to access and manipulate repositories from the tool panel: it is not strictly necessary, but it helps separating the model from the view's actions (i.e. adding or removing a repository).

The Repositories tool also stores and reloads the repositories that the user added when closing and opening the project, respectively. The persistency of the data is taken care of in the REPOSITORIES_STORAGE class, while the storing and loading process take place in the features store_repositories and load_repositories of the EB_WINDOW_MANAGER class.



3.2 Future work

The Repositories tool gives the user the opportunity to inspect and checkout multiple repositories, although it doesn't let the user to directly operate on these repositories. Operations like merging, branching, importing and copying files and folders are often performed directly on the repository and not on the working copy, thus it would be handy to have these features integrated in EVE. The tool panel displays a set of repositories using a tree representation. EB_REPOSITORIES_TREE is the tree widget that displays its children, i.e. instances of EB_REPOSITORIES_TREE_ITEM.



The EB_REPOSITORIES_TREE_ITEM class displays a file or a folder in the repository and is responsible for drawing the contextual menu and executing the actions associated to the menu items. In order to be able to apply a new Subversion command to an item, we need to add a menu item in the contextual menu. There are two steps to follow:

1. Implement the command that you want to perform on the repository's selected item (in the private *implementation* clause of the class);
2. In the context_menu_handler feature, add a new menu item to the contextual menu and associate it to the action implemented in the previous step.

4. Groups tool

4.1 Goal

The classic Groups tool gives a tree representation of all clusters, classes and libraries of the project. Now that we have a Subversion client library written in Eiffel, we have the possibility to add information about the current status of the working copy (if the project is under version control). We also want the user to have the possibility to perform common tasks such as committing, updating, adding or removing files in a fast and unobtrusive way, straight from the Groups tool.

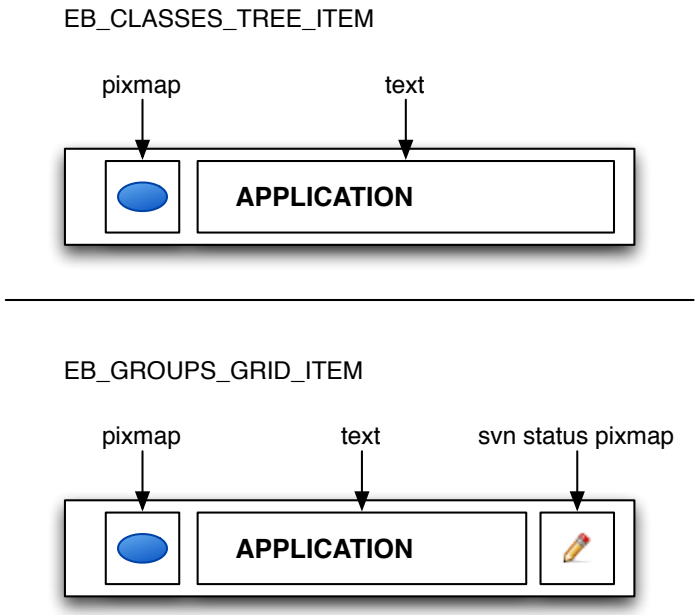
4.2 Architecture

4.2.1 Tool panel

The creation of the tool panel in both the Groups and Repositories tools follows exactly the steps described in the *Tool Integration Development* [1] tutorial and is therefore not covered in this document.

4.2.2 Clusters, classes and libraries tree

Unlike the original Groups tool, the new one is implemented using a tree enabled grid (EV_GRID) rather than an EV_TREE: this migration is required since the latter class does not allow to customize the look of one item. The following picture shows the difference between a row item in the original Groups tool and a row item in the Groups tool that supports version control.



[1] http://dev.eiffel.com/Tool_Integration_Development

The grid class EB_GROUPS_GRID takes care of building and refreshing the tree widget that displays clusters, classes and libraries items, as well as building and drawing the contextual menu for the different grid items.

4.2.3 Grid items

There are four different grid items which are listed and described below:

1. EB_GROUPS_GRID_HEADER_ITEM: the header item is used to group clusters, libraries, assemblies and overrides. This means that in the grid there is a maximum of four header items displayed; empty header items, i.e. with no subitems, are not displayed;
2. EB_GROUPS_GRID_CLASS_ITEM: the class item represents a class in the cluster tree. Information about the status in the working copy, if available, are displayed;
3. EB_GROUPS_GRID_FOLDER_ITEM: the folder item represents a cluster in the cluster tree, including thus all its sub-clusters and classes. It is actually the folder item that recursively creates the tree hierarchy of the clusters group, while the grid only creates the cluster's root.
4. EB_GROUPS_GRID_TARGET_ITEM: a target item.

The Subversion commands are only enabled for classes and folders in the main cluster's item, while libraries and overrides items are not supported.

4.2.4 Contextual menu

The contextual menu of the new Groups tool displays the same options as the old tool (although some of them are not working for the new tool yet) and it additionally offers, at the time of writing, three Subversion client commands:

1. Add: the selected item is put under version control in the working copy;
2. Commit: send the changes of the selected item from the working copy to the repository. A log message can be provided (it is empty by default);
3. Update: bring the changes of the selected item from the repository into the working copy.

If the selected item is a folder, the Subversion command is applied recursively to all their sub-items, exactly as it happens using the command line client. In the next subsection it is explained the necessary process for adding new Subversion commands to the contextual menu.

4.3 Future work

Due to time constraints, only three Subversion commands have been implemented; however, it is very straightforward to add new commands. The steps to follow may vary from two to three, depending on whether optional input arguments are required:

1. In the EB_GROUPS_GRID class, extend the Subversion contextual menu by adding a new menu item in the extend_working_copy_menu feature;
2. If the command requires the user to pass additional parameters, the developer may choose his or her own way to deal with the problem. The solution adopted for the commit command, for example, consists in displaying a small dialog that

allows the user to type a log message. The Subversion dialogs are all organized in one sub-cluster[2];

3. Implement the action that applies the Subversion command to the item. You can refer to the already implemented actions in the *Subversion context menu commands* clause in the EB_GROUPS_GRID class, together with the path_from_pebble feature, that returns the actual name of the selected item and its working path.

Because of the implementation based on a EV_GRID widget, the new Groups tool doesn't have the functionality to restore the expanded and collapsed states of the groups and clusters yet. This causes the tree to collapse all the groups as soon as it is refreshed.

The actions like creating a new cluster or class and locating them in the project files' tree have not been developed for the new panel as they were not in the scope of the project. The intention is however to completely replace, in a near future, the old Groups tool and simplify even more the user interface of EVE.

5. Conclusions

The tool we developed lays the foundations for a new way of working with EVE and Subversion: users don't need to install new third party software, instead they have an integrated instrument built right into the IDE. More importantly, there is no training required in order to learn how to use the tool: all the Subversion commands available are one click away from the Groups tool, with which the user is already familiar with.

This work is to be still considered under development and it is not yet ready for customers use.

---

[2] $EIFFEL_SRC/interface/new_graphical/dialogs/subversion_dialogs/