ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Chair of
Software Engineering

Revision control support for a web-based IDE

Final Report

- **Author:**
  Roland Meyer[1]

- **Supervised by:**
  Martin Nordio
  Christian Estler
  Prof. Dr. Bertrand Meyer

- **Date:**
  Wednesday, January 25, 2012

- **Location:**
  Zürich, Switzerland

[1] E-Mail: *rmy@student.ethz.ch*, Student number: *09 916 131*

Contents

1 Introduction

1.1 Motivation

CloudStudio is a web-based IDE that allows distributed software development. Users can create projects and work on them together with others with the help of several tools that allow communication and project planning. In the current situation, project creation and usage is very limited. Projects can only be created with CloudStudio itself and they cannot be used outside of the framework. It is not possible to e.g. export an existing project such that it can be edited by a different IDE.

1.2 Goal

The goal of this project is to extend CloudStudio with import and export features. It shall be possible for a user to import an existing software project into CloudStudio in addition to creating a new one from scratch. Furthermore the user shall be able to export a project, such that it can be used outside of CloudStudio. Both import and export shall be using SVN, i.e. projects can be imported from or exported into an SVN repository. Additionally, there shall be the possibility to download a project in the format of a zip archive.

1.3 Outline

In chapters 2, 3 and 4, I explain the three parts of this project, namely adding project settings and file system support, as well as features for importing and exporting projects.

In chapter 5 I draw conclusions about the project, explain some limitations and what could be done in the future.

In the appendix chapter A I listed all the changes I made to the database.

2 Project Settings Support

In this section I describe the extensions I made to CloudStudio in order to allow the implementation of the import and export features. For simplicity I will use the term `project` to refer to a software project written in the Eiffel language, as only this kind of project is supported by my implementation. I will use the term `project settings` to refer to a project's main configuration file, i.e. the file with the .ecf extension.

2.1 Project settings

In order for project import to be possible I had to implement various features into CloudStudio first. In the current situation the IDE is very limited in the sense that it is not possible to have subclusters or to make changes to project settings, such as for example adding or removing libraries.

I added a new table to the database called '`project_settings`' (see Chapter A for details). For every project, upon creation or import, a new row is inserted, containing the project id and the content of the project settings file. In the case of an imported project, its project settings are copied unchanged, in the case of a new project, default project settings content is created by the `InitialSourceFilesContents` class which I had to adapt for this. Since project settings are in XML format, they are stored as plain text in the database, ready to be parsed when needed.

The server can access the project settings in a similar way to how he can access other tables. I created a class `ProjectSettingsTable` that extends `DatabaseTable`. It uses the `javax.xml` and the `org.xml` libraries to act as an adapter between the database, where the settings are stored in XML format, and the server with which it interacts using data models. All public methods first fetch the project settings from the database, then parse it, then modify it or extract information from it to return. The `ProjectSettingsTable` class provides methods to modify (i.e. list, insert or remove) a project's libraries, precompiles, clusters as well as a subset of general options. For each of these, I added a model class that can hold its information: `LibraryModel`, `PrecompModel`, `ClusterModel` and `Project-SettingsModel`. Additionally there are features that return and edit the project settings as a string, allowing arbitrary changes to the project settings, which I will explain later.

The client can use most of these methods through RPC calls to the newly added service called `Pro-jectSettingsService`. I will explain in section 2.3 how they are used.

2.2 File system

With the addition of allowing clusters and subclusters it was necessary to extend CloudStudio's file system. In the already existing source files table there is a field called '`folder_id`' which was not used by the system so far. In my solution, this field is now used to point to the folder in which the file is located. I added a table called `` `folders` `` to store folder information such as the folder's id, name, its parent folder's id and the id of the project to which it belongs. Every project has an implicit root folder with id 0. This root folder is never stored in the database but files and folders may point to it as their parent directory.

Having these extensions it was possible to add support for clusters. If a new cluster is created from the `ProjectSettingsService`, the system will first try to create the corresponding folder for it. Creating a cluster fails if a folder with the same name already exists. This is reasonable because it is defined by the Eiffel language that a cluster name must be unique within a project.

Because one goal of my project is to allow importing of arbitrary Eiffel projects, it was necessary to extend the file system further to not only support source files and folders but also byte encoded files, e.g. images or audio files. For simplicity I will refer to these as `binary files`. To store such files I added another table called `` `binary_files` ``. Like the existing `` `source_files` `` table it has fields for its id, name, project name, folder id and content. The content field is a `MEDIUMBLOB` which allows files of size up to 16 MB. In analogy to source files, binary files are represented by `BinaryFileModel`s and `Bina-ryFileMiniModel`s and there are methods to add or delete them in the `BinaryFilesTable` class. It is important to note that `BinaryFileModels` are *not serializable* and thus the content of a binary file can only be used by the server and not the client. This implies that the user can only delete binary files from a project, and the only way that binary files can be inserted into a project is if they are imported together with an entire project that contains them.

2.3 Changes to the GUI

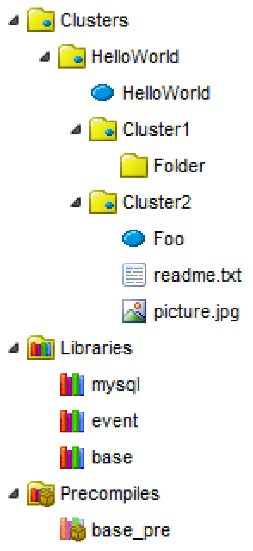


Image 1: The Groups Tool

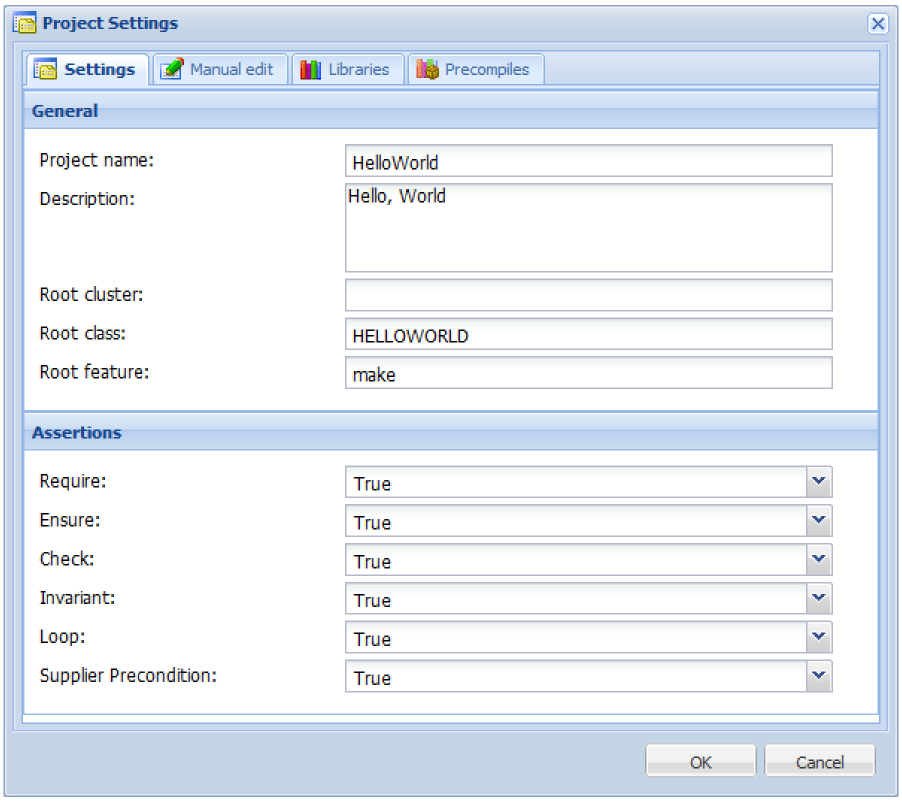
The changes made to the file system representation and the storing of project settings is also represented in the GUI. In the main view of the IDE, there is a panel to the right labelled 'Groups'. Before, it only listed the classes in a project and indicated that the project uses the `base` library. The only action the user could execute was open class files and create new ones in the root directory. With the changes made to the file system and the way project settings are stored I had to make many changes to the groups tool.

Image 1 shows the groups tool of an example project. The groups tool contains a tree structure (provided by the GWT classes `TreeStore` and `TreePanel`) with three root items; `Clusters`, `Libraries` and `Precompiles`. It now uses an actual RPC call to the server, using the `ProjectSettingsService`, to fetch a list of all the libraries in the project. The libraries are then added to the tree structure under the root item `Libraries` (as can be seen on the image). The precompiles are retrieved and displayed in a similar way.

A representation of the project's file system is added under the `Clusters` item. The goal here was to make it behave in a similar way to the groups tool in EiffelStudio. There are four kinds of items that can be represented; Clusters, folders, (class) files and binary files. My first approach was to use different RPC calls to retrieve those and put the tree together on the client side, but it turned out to be an enormous workload for the client for large projects. So I shifted this task to the server. I implemented a method called `getClusterTree` in the `ProjectSettingsService` which builds up the tree structure on the server side, then sends it as a list of `ModelData` items in depth-first pre-order to the client, where a single loop is enough to transform the list back into a tree. This is possible because all items in the list contain information about their parent item, namely they contain the id of the parent folder. For clusters, the name of the parent cluster is used instead since clusters don't have an id but their names are guaranteed to be unique. Clusters at the root have an empty string for their parent cluster name. To indicate the different file types to the users, different icons are used for binary files, class files (ending in .e) and other files, e.g. text files (see Image 1 for an example).

I also extended the groups tool's context menu. Users can now not only create new classes as before but also add clusters. These new items are added where the user opened the context menu if applicable and otherwise they are inserted at the root. Users can also delete items this way, including libraries and precompiles. Folders and clusters can only be removed if they are empty. All of these actions use the features provided by the `ProjectSettingsService` class, except for the adding and removing of files which use the `FileService` class.

To allow the user to make changes to the project settings I added a button to the toolbar at the top of the screen labelled `Project Settings`. Clicking it opens a dialog window with four tabs; 'Settings', 'Manual edit', 'Libraries' and 'Precompiles' (see Image 2). In the 'Libraries' and 'Precompiles' tabs the user sees a list of the libraries and precompiles currently in the project and is provided with the option to remove them or to add new ones. If he clicks on the `Add libraries` (respectively `Add precompiles`) another dialog window opens showing a list of all libraries (precompiles) that are supported by the server and that are not in the project already (or are in the project but were deleted in the previous dialog). This list is retrieved through an RPC call to the newly added `LibraryService` which takes the list from the database table `supported_libraries` (`supported_precompiles`) (see Chapter A for details).



General

Project name:	HelloWorld
Description:	Hello, World
Root cluster:	
Root class:	HELLOWORLD
Root feature:	make

Assertions

Require:	True
Ensure:	True
Check:	True
Invariant:	True
Loop:	True
Supplier Precondition:	True

Image 2: The project settings dialog

In the `Settings` tab the user has the option to change basic project settings such as project name, description, project root and assertions. Note that changing the project name or description does not affect the respective fields in the `project` table but only the actual project settings in the `project_settings` table. If the user clicks the dialog's `OK` button the changed values are packed into a `ProjectSettingsModel` and sent to the server through the `ProjectSettingsService`. There the original XML content of the project settings is parsed, the changed values are inserted and the XML content is stored back into the database.

In the `Manual edit` tab the user has access to the actual content of the project settings file. This way he can make changes which are not possible through the basic options in the `Settings` tab. As soon as the user makes changes in this view, the other three tabs are disabled to avoid inconsistencies. The same thing occurs vice-versa. The user must make sure to maintain correct XML syntax and to submit valid Eiffel project settings. If problems arise while parsing the project settings, the `ProjectSettingsService` returns empty results for all requests, making it impossible to access project files and folders.

3 Project Import

In the project view where the user can see the different projects he has access to I added two buttons. One of these buttons is labelled 'Import project' and clicking it will open a new dialog. This dialog looks similar to the existing dialog to create new projects, but there are additional fields to enter the URL of an SVN repository, a revision number and user credentials for the repository. Other than in the 'New project' dialog the 'Project Name' field is optional here. If it is left empty the project name will be taken from the imported project.

Once the user starts to import a project a wait dialog is displayed and an RPC call through the `ProjectService` is started. The project import goes step by step using classes and features from the SVNKit library (svnkit.com). The server tries to connect to the SVN repository; if necessary it will send the credentials as well. If the given repository URL points to a project settings file, i.e. the URL ends in '.ecf' the server will try to fetch that file. Otherwise it will scan the root directory and takes the first project settings file it encounters. The content of the project settings file is copied from the repository into a new entry in the `project_settings` table. If no project name was provided, the server will now try to extract it from the copied settings. After that all remaining files and folders are copied recursively from the repository into the database. Binary files which are too large (see Chapter A) are not copied. If anything goes wrong during import, e.g. the credentials were wrong, no project settings file was found or there already existed a project with the same name, an exception is thrown and passed back to the user where it is displayed as an error message. In this case all changes to the database are undone. If the import was successful, a message is displayed to the user which also contains the number of binary files omitted, if any. After that the imported project appears in the list of projects and is ready to be used.

4 Project Export

The other button added to the projects view is labelled `Export project`. The user can select a project from the list and click this button to open the export dialog. He can choose between two types of export: `SVN` and `Zip Archive`.

4.1 SVN Export

If the user picks `SVN` as the export type he can enter a URL to an SVN repository, its credentials and a log message. Once he clicks the `OK` button an RPC call to the `ProjectServ`ice starts the export process and like for the import feature a wait dialog is displayed to the user. In the first step, the server writes all files and folders, including binary files, from the database to its own hard drive. This is done using the `prepareProjectCompile` method which is also used for compiling projects. I had to extend this feature to support the additions made to the file system, such as folders, binary files and the project settings file itself. Once all files are written the server tries to connect to the SVN repository, again using the SVNKit library and providing credentials if necessary. It will then check if the repository is empty. My implementation currently requires the target repository to be empty because merging and comparing different versions of files is not supported. If the repository is not empty exporting is aborted and the user is informed by an error message.

At this point, all the project's files and folders are recursively copied to the repository, using an instance of the `ISVNEditor` class from the SVNKit library. Like for the import, if anything goes wrong during export the entire process is aborted and the changes made to the repository are rolled back. If the export was successful, the user is informed by a message.

4.2 Archive Export

If the user picks `Zip Archive` as the export type he does not need to provide any parameters. Clicking the `OK` button will trigger an RPC call to `ProjectServ`ice and the exporting process begins on the server. The first step is the same as for the SVN export; all the project's files and folders are written to the hard drive. The server then makes sure that the target folder to store the zip archive exists. Its location is defined by the `CloudStudioConfiguration` class and can be retrieved using the method `getExportedArchivesHDDPath`. If the folder exists, any existing archive for the project is removed. The name for the new zip archive consists of the project id, to make sure that there is at most one archive per project, followed by a capital U and the user's id, followed by a capital R and random integer of up to 6 digits. The project files are recursively copied into this archive using the java.util.zip library. Once the archive is complete, the randomly generated file name is passed back to the client and a URL is displayed where the user can download the archive from.

The URL points to a special servlet, named `DownloadServlet.` It extracts the archive's file name from the URL, opens the archive from the archive directory and sends it to the client as a HTTP response. After the archive has been transferred it is deleted on the server to prevent unauthorized users to download the project by guessing the URL.

5 Conclusions

The newly added features greatly improve the possibilities of working with CloudStudio. Existing Eiffel projects can easily be imported from an SVN repository with just two button clicks and exporting from CloudStudio to a repository is just as easy. Thanks to the additions to the groups tool and the file system, the user can work on more advanced projects with multiple clusters, folders, classes and arbitrary data files. No additional training is required for users as the new groups tool can be operated in a similar way as one does in EiffelStudio, with which the user is already familiar.

In the future, it might be necessary to change the way in which files are written from the database to the hard drive. In my implementation, the system does not check which files are already written and which aren't. This may cause great delays when compiling or exporting large projects.

The ultimate goal of this project was to import an arbitrary Eiffel project, modify it and export it back to a repository. This was successfully done with the DOSE2011 project.

A Database changes

This section contains a list of the tables I added to the database for the project.
In order for the import of binary files to work properly, it is important that the `max_allowed_packet` parameter of the system's MySQL database is set to a value of at least 16 MB. Otherwise large binary files are omitted on import. Files larger than 16 MB are never imported as they will not fit into the database.

binary_files
Used for storing binary files (byte encoded files) in a project.

id	INT(11)	The id of the binary file
project_id	INT(11)	The id of the project to which the file belongs to
folder_id	INT(11)	The id of the folder in which the file is located (0 if it is in the root directory)
name	VARCHAR(100)	The name of the binary file
content	MEDIUMBLOB	The content of the file

folders
Used for storing folders in a project. The root folder (id 0) is never stored.

id	INT(11)	The id of the folder
name	VARCHAR(100)	The name of the folder
parent	INT(11)	The id of the parent folder (0 if it is in the root directory)
project_id	INT(11)	The id of the project to which the folder belongs to

project_settings
Used for storing the content of a project's settings file.

project_id	INT(11)	The id of the project to which the settings belong to
content	LONGTEXT	The text content of the project settings file

supported_libraries
Used to provide a list of all libraries that are supported by the system to the user. This table should at least contain an entry for the Eiffel base library.

id	INT(11)	The id of the library
name	VARCHAR(60)	The name of the library
location	VARCHAR(120)	The path in the server's file system where the library is located. The path should be system independent, i.e. should start with '$ISE_LIBRARY'.

supported_precomps
Used to provide a list of all precompiled libraries that are supported by the system to the user. This table should at least contain an entry for the precompiled Eiffel base library.

id	INT(11)	The id of the precompiled library
name	VARCHAR(60)	The name of the precompiled library
location	VARCHAR(120)	The path in the server's file system where the precompiled library is located. The path should be system independent, i.e. should start with '$ISE_PRECOMP'.